AFML-TR-70-78

UR-70-0167

COMPUTER-AIDED DESIGN *for* NUMERICALLY CONTROLLED PRODUCTION

J. E. Ward

AM027431

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TECHNICAL REPORT AFML-TR-70-78
June 1970

FABRICATION BRANCH
MANUFACTURING TECHNOLOGY DIVISION
AIR FORCE MATERIALS LABORATORY

Air Force Systems Command
Wright-Patterson Air Force Base, Ohio

Private STINET

Home | Collections
View Saved Searches | View Shopping Cart | View Orders

Add to Shopping Cart

Other items on page 1 of your search results: **1**

View XML

Citation Format: Full Citation (1F)

Accession Number:
AD0872259
Citation Status:
Active
Citation Classification:
Unclassified
Fields and Groups:
120500 - Computer Programming and Software
120600 - Computer Hardware
130800 - Mfg & Industrial Eng & Control of Product Sys
Corporate Author:
MASSACHUSETTS INST OF TECH CAMBRIDGE ELECTRONIC SYSTEMS LAB
Unclassified Title:
(U) Computer-Aided Design for Numerically Controlled Production.
Title Classification:
Unclassified
Descriptive Note:
Final technical rept. 1 May 67-30 Jan 70,
Personal Author(s):
Ward, John E
Report Date:
Jun 1970
Media Count:
134 Page(s)
Cost:
$14.60
Contract Number:
F33615-67-C-1530
F33615-69-C-1341
Report Number(s):
ESL-FR-420
AFML-TR-70-78
Project Number:
AF-863-7
Monitor Acronym:
AFML
Monitor Series:
TR-70-78
Report Classification:
Unclassified
Descriptors:
(U) (*PRODUCTION CONTROL, AUTOMATION), (*COMPUTER PROGRAMMING, PROBLEM

SOLVING), GRAPHICS, PROGRAMMING LANGUAGES, DISPLAY SYSTEMS, COMPILERS

Identifiers:

(U) AED-0 PROGRAMMING LANGUAGE, *COMPUTER AIDED DESIGN, COMPUTERS, GRAPHICS, *NUMERICAL CONTROL.

Identifier Classification:

Unclassified

Abstract:

(U) The report summarizes the activities of the M.I.T. Computer-Aided Design (CAD) Project in the final implementation phase of a generalized 'system of software systems' for generating specialized problem-oriented man-machine problem-solving systems. Known as the AED approach (for Automated Engineering Design) the Project results are applicable not only to mechanical design, but to arbitrary scientific, engineering, management, and production system problems as well. Program accomplishments are supported by hardware and software innovations in computer graphics. All results have been programmed using machine-independent techniques in the Project's AED-0 Language, based on Algol-60. (Author)

Abstract Classification:

Unclassified

Distribution Limitation(s):

01 - APPROVED FOR PUBLIC RELEASE

Source Serial:

F

Source Code:

127200

Document Location:

DTIC AND NTIS

Change Authority:

ST-A USAFSC, LTR, 26 MAY 72

Distribution Change Order
Refer to Change Authority Field

Private STINET

Home | Collections
View Saved Searches | View Shopping Cart | View Orders

Other items on page 1 of your search results: 1 | 2 | 3 | 4 | 5 | 6

Citation Format: Full Citation (1F)

Accession Number:
AD0872259
Citation Status:
Active
Citation Classification:
Unclassified
Field(s) & Group(s):
120500 - COMPUTER PROGRAMMING AND SOFTWARE
120600 - COMPUTER HARDWARE
130800 - MFG & INDUSTRIAL ENG & CONTROL OF PRODUCT SYS
Corporate Author:
MASSACHUSETTS INST OF TECH CAMBRIDGE ELECTRONIC SYSTEMS LAB
Unclassified Title:
(U) **Computer-Aided Design** for Numerically Controlled Production.
Title Classification:
Unclassified
Descriptive Note:
Final technical rept. 1 May 67-30 Jan 70,
Personal Author(s):
Ward, John E.
Report Date:
Jun 1970
Media Count:
134 Page(s)
Cost:
$14.60
Contract Number:
F33615-67-C-1530
Contract Number:
F33615-69-C-1341
Report Number(s):
ESL-FR-420
AFML-TR-70-78
Project Number:
AF-863-7
Project Number:
AF-86309
Monitor Acronym:
AFML
Monitor Series:
TR-70-78
Report Classification:
Unclassified
Descriptors:
(U) (*PRODUCTION CONTROL, (*COMPUTER PROGRAMMING, AUTOMATION), PROBLEM SOLVING), GRAPHICS, PROGRAMMING LANGUAGES, DISPLAY SYSTEMS, COMPILERS
Identifiers:
(U) AED-0 PROGRAMMING LANGUAGE, *COMPUTER AIDED DESIGN, COMPUTERS, GRAPHICS, *NUMERICAL CONTROL,

Identifier Classification:
Unclassified

Abstract:
(U) The report summarizes the activities of the M.I.T. Computer-Aided Design (CAD) Project in the final implementation phase of a generalized 'system of software systems' for generating specialized problem-oriented man-machine problem-solving systems. Known as the AED approach (for Automated Engineering Design) the Project results are applicable not only to mechanical design, but to arbitrary scientific, engineering, management, and production system problems as well. Program accomplishments are supported by hardware and software innovations in computer graphics. All results have been programmed using machine-independent techniques in the Project's AED-0 Language, based on Algol-60. (Author)

Abstract Classification:
Unclassified

Distribution Limitation(s):
01 - APPROVED FOR PUBLIC RELEASE

Source Serial:
F

Source Code:
127200

Document Location:
DTIC

Change Authority:
ST-A USAFSC, LTR, 26 MAY 72

COMPUTER-AIDED DESIGN FOR NUMERICALLY CONTROLLED PRODUCTION

Final Technical Report

1 May 1967 - 30 January 1970

John E. Ward

Electronic Systems Laboratory
Electrical Engineering Department
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Cambridge, Massachusetts 02139

Contracts

F33615-67-C-1530
F33615-69-C-1341

FOREWORD

This Final Technical Report submitted in March 1970, summarizes the work performed from 1 May 1967 through 30 January 1970 under two successive United States Air Force Contracts: F33615-67-C-1530 from 1 May 1967 to 30 November 1968; and F33615-69-C-1341 from 1 December 1968 to 30 January 1970. Details of the work have previously been recorded in a series of Technical Reports on specific subjects. Prior work from 1 December 1959 through 3 May 1967 was recorded in Final Technical Report AFML-TR-68-206, May 1968.

The M.I.T. number assigned to this report is ESL-FR-420.

Contracts F33615-67-C-1530 and F33615-69-C-1341 with the Electronic Systems Laboratory of Massachusetts Institute of Technology, Cambridge, Massachusetts, were initiated under Manufacturing Methods Projects 863-7 and 86309, "Integration of Design Data into Numerical Control." They were accomplished under the technical direction of Mr. W. M. Webster, Mr. M. A. Guenther, and Lt. R. Coe, Frabrication Branch, MATF, Manufacturing Technology Division, Air Force Materials Laboratory, Wright-Patterson Air Force Base, Ohio.

The program covered in this report was the result of the efforts of many people over an extended period — May 1, 1967 to January 30, 1970. Listed below are the 50 technical personnel (faculty, staff members, visiting staff, graduate and undergraduate students) who participated directly in the work during this period, organized according to the subgroupings within the Computer-Aided Design Project.

Mr. Douglas T. Ross has served as Project Engineer for the Computer-Aided Design Project for this entire period. The support and counsel of Professor J. Francis Reintjes, Director of the Electronic Systems Laboratory, is gratefully acknowledged.

From the Computer Applications Group, Electronic Systems Laboratory

Douglas T. Ross, Group Leader and Project Engineer
Clarence G. Feldmann, Associate Leader
Dr. Jorge E. Rodriquez, Assistant Leader

Eric C. Anderson
Henry G. Baker, Jr.
Reuben J. Bigelow
Richard H. Bryan
Frederick Cioramaglia
Dr. Ronald W. Cornew
Robert S. Eanes
N. Dudley Fulton
Irene G. Greif
William M. Inglis
Peter Johansen
Panos Z. Marmarelis
Robert C. Nelson
Robert B. Polansky
John R. Ross
Daniel E. Thornhill
John F. Walsh
John E. Ward
Thomas S. Weston
Barry L. Wolman
Robert V. Zara

Visting Staff of the AED Cooperative Program

Frank Bates	- Union Carbide Company
Donald J. Cameron	- Ferranti Limited
John T. Doherty	- Raytheon Manufacturing Company
Richard B. Gluckstern	- Univac Div. Sperry Rand Corporation
G. Lawrence Lane	- Sandia Corporation
Robert J. McDowell	- Honeywell EDP Division
Richard A. Meyer	- IBM Corporation
Arthur T. Nagai	- The Boeing Company
Irwin Wenger	- Raytheon Manufacturing Company
Stephan Zurnaciyan	- Northrop Corporation

From the Display Group, Electronic Systems Laboratory

John E. Ward, Group Leader and Deputy Director,
Electronic Systems Laboratory

Robert H. Stotz, Assistant Group Leader

Robert J. Ascott
Abhay K. Bhushan
Michael F. Brescia
David G. Chapman
Thomas B. Cheek
James G. Fiasconaro
Frank B. Hills
William Hutchison
Robert G. Rausch
Christopher L. Reeve
Thomas L. Smith
W. David Stratton
Jonathan R. Sussman
Daniel E. Thornhill
Dietrich Vedder
Albert Vezza

This project has been accomplished as a part of the Air Force Manufacturing Methods Program, the primary objective of which is to develop, in a timely basis, manufacturing processes, techniques, and equipment for use in economical production of USAF materials and components.

Your comments are solicited on the potential utilization of the information contained herein as applied to your present and/or future production programs. Suggestions concerning additional manufacturing methods development required on this or other subjects will be appreciated.

This technical report has been reviewed and is approved.

[illegible]

JACK R. MARSH
Chief, Fabrication Branch
Manufacturing Technology Divisin

ABSTRACT

This report summarizes the activities of the M.I.T. Computer-Aided Design (CAD) Project from 1 May 1967 through 30 January 1970 in the final implementation phase of a generalized "system of software systems" for generating specialized problem-oriented man-machine problem-solving systems using high-level language techniques and advanced computer graphics. Known as the AED Approach (for Automated Engineering Design) the Project results are applicable not only to mechanical design, as an extension of earlier development of the APT System for numerical control, but to arbitrary scientific, engineering, management, and production system problems as well. As described in prior report AFML-TR-68-206, May, 1968, advanced techniques for verbal and graphical language and generalized problem-modeling are based on the concept of a "plex" which combines data, structure, and algorithmic aspects to provide complete and elegant representation of arbitrary problems. Program accomplishments are supported by hardware and software innovations in computer graphics. All results have been programmed using machine-independent techniques in the Project's AED-0 Language, based on Algol-60.

During this concluding 32-month phase of the 10-year CAD program at M.I.T., the major emphasis has been on bootstrapping the AED systems to third-generation computers. A series of field-trial systems was made available to industry, culminating in July 1969 in formal release of Version 3 of AED for IBM 360-series computers in both batch and time sharing, and partial completion of a compatible version for the Univac 1108 computer. Report topics include: The bootstrapping process; user documentation for the AED system; several application studies to demonstrate use of AED techniques in language design, system building, and computer graphics; and Project interaction with industry.

ACKNOWLEDGEMENT

The source material for the preparation of this comprehensive final report on project activities was drawn from various existing project publications and progress reports plus previously unpublished memoranda and some new material. The author would like to acknowledge the contributions of D. T. Ross in Chapter I, of C. G. Feldmann in Chapters II and III, and of R. B. Polansky and R. W. Cornew in Chapter V.

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction of this report, in whole or in part is permitted for any purpose of the United States Government.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

This report covers the activities of the MIT Computer-Aided Design Project from 1 May 1967 through 30 January 1969, the concluding phase of a fruitful 20-year investigation into automated manufacturing for the Air Force. In the period from 1949 - 1955, the first numerically controlled machine tool was developed and demonstrated, and in the period from 1955 - 1959, the APT (Automatically Programmed Tool) system was developed in cooperation with the Aerospace Industries Association. Project attention since 1959 has been devoted to the subject of computer-aided design, with two major lines of investigation: techniques for the "automation" of software construction to provide the groundwork necessary for flexible and economical application of computers in the design process; and techniques and equipment for man-machine communication. Of course, there were many subsidiary and supporting lines of inquiry over this 10-year span, but the bulk of these have already been reported in a previous final report for the period 1 December 1959 to 3 May 1967,* and many separate technical reports on specific subjects.

During the period covered by this report, the emphasis was on "reduction-to-practice", i.e., putting the programs and techniques that had been established into a form suitable for distribution to the aerospace industry. The major task was to convert the AED (<u>A</u>utomated <u>E</u>ngineering <u>D</u>esign) system of programs developed on second-generation computer facilities at M.I.T. to the third-generation computers now widely used in industry. This has resulted in formal release of AED systems for the IBM 360 and Univac 1108 computers. The process by which this was done is part of the AED story; the achievement of a remarkable degree of machine independence, which

---

* Report AFML-TR-68-206 (see report list for complete citation).

provides a protection against the perennial problem of software obsolescence as computer technology changes. This process, called "bootstrapping", is reviewed briefly in Section B of this Introduction, and details of the actual bootstrapping of AED to the IBM 360, resulting in the final released system, are presented in Chapters II and III. Prior to this, however, it is useful to review in Section A the historical goals of the investigations which have led to AED. Much of Section A has been adapted from a similar section of the previous Final Report AFML-TR-68-206.

A. REVIEW OF PROJECT GOALS

Work on computer-aided design within the Computer Applications Group of the Electronic Systems Laboratory at M.I.T. predates the contractual periods covered by this report. The real beginning was in June of 1956 when the group inherited the M.I.T. activity in numerical control from the preceding project, which had been primarily engineering oriented, and focused on automatic programming for numerical control through the APT Project. In the very first interim report of the APT Project, brief reference is made to the ultimate extension of the preparation of numerical control information into the design area. Throughout the APT System development, this ill-formed concept always was in the background. As the work on the development of the APT System was drawing to its logical conclusion with the cooperative effort with companies in the Aerospace Industries Association, the possibilities of extending the sophisticated use of computers into the design preparation stages which precede part programming came more sharply into focus. Since the early fifties we had made extensive use of on-line displays and manual intervention (the first graphical input through a display was done in 1954) and had developed a backlog of sophisticated computer usage techniques in various areas. Some of this experience had been used profitably in the APT System development, but other portions needed a broader context in order to be brought to fruition. These experiences, coupled with an informal acquaintanceship with the problems of aerospace design and manufacturing gained from the intimate contact with aerospace companies during the APT effort, formed the background for the initial proposal that useful results could be expected from a program in computer-aided design.

Even our incomplete knowledge of the aerospace design-to-manufacturing cycle was sufficient to demonstrate clearly that a small academic project could not hope to create a system which would solve everybody's problem directly. Furthermore, it was felt that it would not be appropriate to concentrate on a limited and specialized application which would be of direct utility to only one or a few of the potential companies we hoped to benefit. Instead, we felt that the most appropriate program would be one aimed at fundamentals, with close liaison with industry once preliminary useable results had been obtained. Industry, especially in aerospace matters, had historically been project oriented, with insufficient time for a truly long-range technological effort such as we felt was needed. On the other hand, the ability of the aerospace industry to adapt rapidly to sophisticated technological innovations is well known; the rapid incorporation of numerical control being but one of many excellent examples. We hoped, therefore, to be able to establish a systematized solution to some of the basic problems of generalized computer-aided design, and then to serve a catalytic role in stimulating industry itself in the application of the results to diverse application areas.

Our deliberations on the role which we could play began even before we had coined the term "computer-aided design" in the late 1950's, and, in fact, a part of the early discussions centered around the distinction between "automatic" and "computer-aided" design and "computer aids to" design. It seemed quite clear to us that any work in automatic design, in which a parameterized design is optimized in some fashion by parameter variation, would of necessity be too clearly related to a particular application area to permit a general approach. Although various aspects of the APT System analyses and portions of earlier work in which we had used various linearization and adaptive programming techniques were related to automatic design, we felt that such questions definitely should occupy a secondary role. Similarly, we were not interested in advancing the use of computers as aids to existing design practice. Instead, we wanted to couple a man and a machine into a problem-solving team suitable for fresh design problems requiring creative solutions, and performing better than either man or machine alone. Our ultimate goal was to have a system capable of going from the conception of the need for a part through to the finished product by means of numerical control.

We sought to establish a system whose language and characteristics could be adapted to meet the needs of the user, whatever the domain of application. Our view was that it was impossible to put bounds upon the problems with which the system would have to cope, because design itself knows no bounds. Geometry, materials, aerodynamics, thermodynamics, and even aesthetics, all may play determining roles in a given design. If the system was truly to be applicable to creative design, it would have to be adaptable to these and many other areas. Thus in our view, there was no distinction between computer-aided design and generalized man-machine problem-solving.

These grandiose plans were not entirely wishful thinking, for the previous activities of the Computer Applications Group had touched upon many of the requisite areas and it seemed clear that even modest success would have handsome payoff if properly applied. Many of the basic ideas on which we hoped to base the new program emerged and went through their earliest refinements in the context of our final contributions to the APT System. Still others were exercised as part of several small investigations which we carried out at the close of the APT Project in preparation for the new computer-aided design focus.

As our viewpoint has sharpened and as our techniques and general approach have matured over the years, we have changed terminology slightly, so that instead of speaking of adapting a general system to a special purpose, we now prefer to think of employing a family of generalized systems to create a specialized new member of the family. We call the family of systems by the general name AED (an acronym for Automated Engineering Design) and tout "the AED Approach" as the entire sweep of concepts, techniques, and working tools for creating specialized computer-aided design systems.

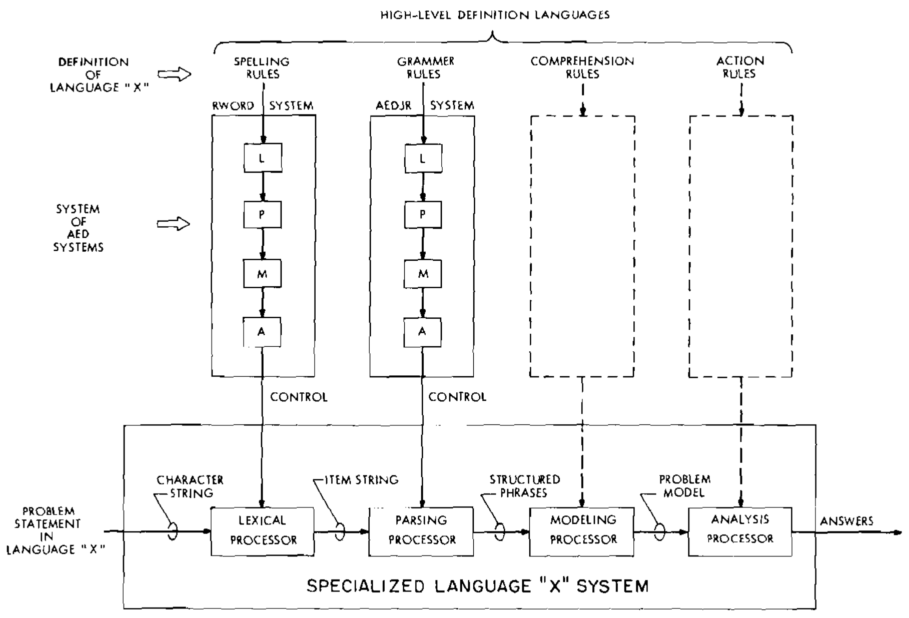
The terminology and techniques are refined, but the overall concept is the same. The AED Approach to system building is illustrated in Fig. 1.

B. THE AED BOOTSTRAPPING PROCESS

The AED Compiler is composed almost entirely of programs written in the AED-0 Language. The remaining portion is a minimum number of additional, machine-language programs which handle a few operations that are totally machine dependent and serve to interface the AED Compiler with the machine environment. Given the AED System operating on one computer, called the Host Computer, the programs which constitute AED can be compiled into a new form which will operate on another computer, called the Target Computer. This technique called "bootstrapping", is briefly summarized below to give a background for the work described in Chapters II and III.

Most of the AED-0 Language source programs are completely machine-independent, i.e., they deal with operations that must be performed on any computer. Machine-dependent data structures (called "state beads") describing the Target machine are all defined in a single program segment that is inserted in the machine-independent programs by an .INSERT statement. A smaller number of the compiler's AED-0 Language source programs deal with machine-dependent aspects of the compilation process. The number of these programs depends upon the special machine characteristics and how "smart" the compiler is in the use of these characteristics.

In addition to the body of the compiler programs, there is a series of packages of procedures which we call the "Support Package". Each portion of the Support Package handles one aspect of the system-building process (dynamic storage allocation, string manipulation, etc.), and the compiler's source programs make frequent use of these packages. In the same way as the compiler source programs, the Support Package is divided into AED-0 Language source programs, both machine independent and machine dependent, plus a few machine-language programs.



This figure indicates the AED approach to system building. Any software system for solving a problem can be broken down into four phases: a lexical processor which forms items (words) from the individual characters in the input string of the language being used, a parsing processor which properly groups the items according to the grammatical rules for the language, a modeling processor which extracts the meaning and sets up a model suitable for computer manipulation, and an analysis processor which carries out the desired solution. The AED-1 Compiler is an example of such a system in which the problem language is AED-0 (based on ALGOL-60) and the "answers" are compiled (object) programs.

Two of the system-building programs have been completed -- the RWORD (read-a-word) System which uses the same four steps to produce a lexical processor from high-level language descriptions, and the AEDJR System which builds a parsing processor from high-level command-level descriptions of the metalinguistic properties of a new language. A feature of all these systems is that they are all written in AED-0, which permits a process called "bootstrapping" to be used in transferring the system to different computers.

Fig. 1 The AED Approach to System Building

1. Graphical Notation

The bootstrapping process is most easily described in terms of a pictorial representation which contains all of the relevant information in a compact form. In this representation, the "triangle" symbols shown in Fig. 2 are used to represent types of program. The current

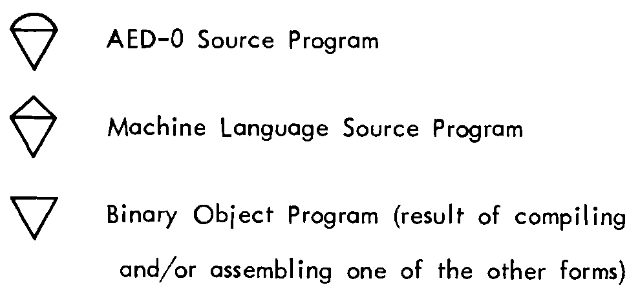

AED-0 Source Program
Machine Language Source Program
Binary Object Program (result of compiling and/or assembling one of the other forms)

Fig. 2 Graphical Conventions for Program Type

AED Compiler generates character-string output for a format accepted by the existing machine-language Assembler supplied with the Target computer (which may be the Host Computer). In this way, detailed concern about binary bit patterns was avoided for the initial bootstrap. Later, similar techniques may be used to eliminate the Assembler step by producing bit strings directly instead of character strings. At present, however, the compilation of an AED-0 Language source program into binary object program is a two-stage process as shown in Fig. 3. The compiler and assembler may operate on the same, or on different computers.

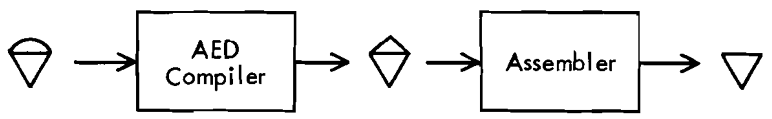


Fig. 3 The AED Compilation Process

Several further sets of conventions need to be established. One of these is that if a program contains machine-dependent statements, we write the letter H or T inside the triangle to indicate Host or Target dependence.

Triangle representation of the six categories of programs of an AED Compiler creates a hexagon, where the orientation of the triangles within the hexagon depicts each program's category. The Support Package occupies the bottom half and the Compiler the top half. AED-0 source programs are shown in the left-side (machine-dependent) and middle (machine-independent) sectors and machine-language programs occupy the right-side sectors. The sectors for the graphical notation may thus be identified as shown in Fig. 4.

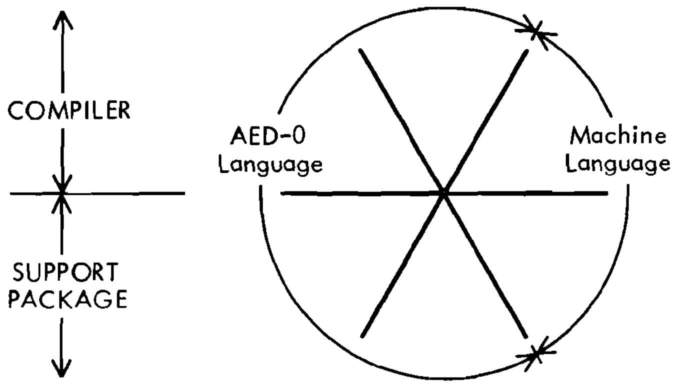


Fig. 4 Hexagon Representation of Program Categories

Using the conventions of Fig. 2 for program type, adding the H and T symbols to show machine dependence, and showing the environment via a box around the hexagon, the complete representation for a Host compiler is obtained as shown in Fig. 5.

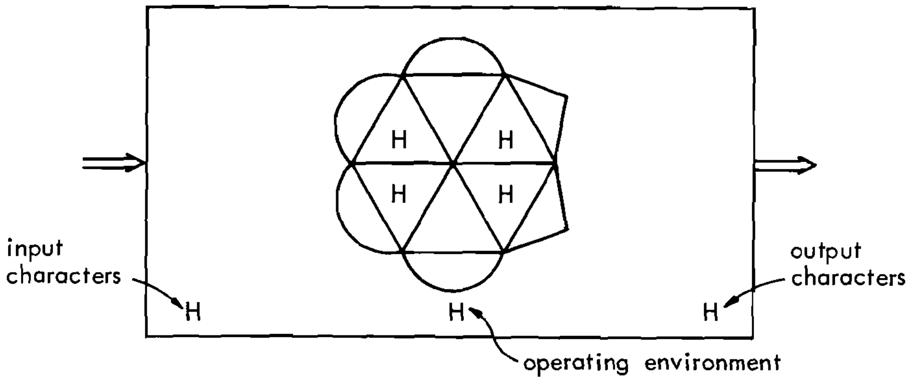


Fig. 5 Complete Representation for a Host Compiler

2. Representation of the Overall Bootstrap Process

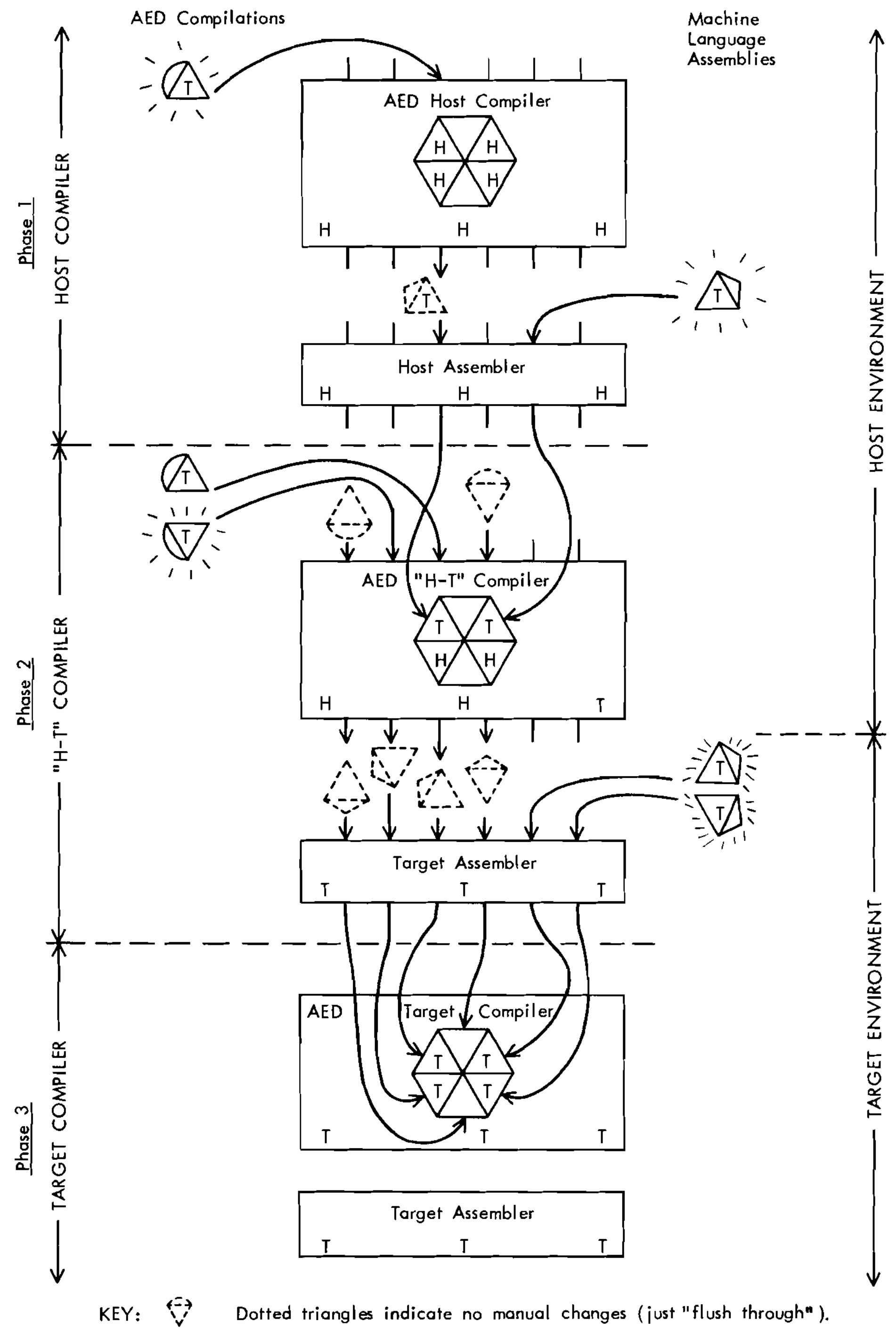
Using the graphical conventions that have been established above, the complete process of bootstrapping AED from one computer to another is shown in Fig. 6. As indicated, this is a three-phase process. The first phase produces portions of a new host-machine compiler, that with additional steps in the second phase becomes an "H - T" (half-bootstrap) compiler that operates on the host computer, but produces assembly language for the target computer that may then be assembled and run on the target computer. The final phase is to use the H - T compiler to transfer the AED compiler itself to the target computer.

The process shown in Fig. 6 is rather complex. To complete this brief summary of the process, it is instructive to select one triangle and follow its course through the diagram. Figure 7 shows the path of the machine-dependent AED-0 portion of the compiler from the original host version to the final target version. We see that steps 1 through 4 in Fig. 7 prepare a compiler which accepts AED-0 programs in the Host environment and produces output for the Target machine (called the "H - T" compiler). Processing AED-0 programs through this H - T compiler then produces pieces of the desired Target Compiler.

3. Relative Program Sizes

Up to now, nothing specific has been said about relative sizes of the six hexagon pieces, and all six have been drawn the same size. To give a better feel for the extent of true machine independence which has been achieved in the AED Compiler, and of the reprogramming job involved in the process, data for the May, 1968 bootstrap to the IBM 360 Computer are shown in Fig. 8. The number of 32-bit binary machine words for each of the six sectors of the Target Compiler is shown by a proportionally-drawn arrow through the sector.

Figure 9 shows a breakdown based on programs and tables. This distinction is of interest since tables are set up with little new thinking, whereas programs require creative decisions on the part of the programmer. The machine-independent AED-0 portion requires no reprogramming except for a possible redesign of the state beads.

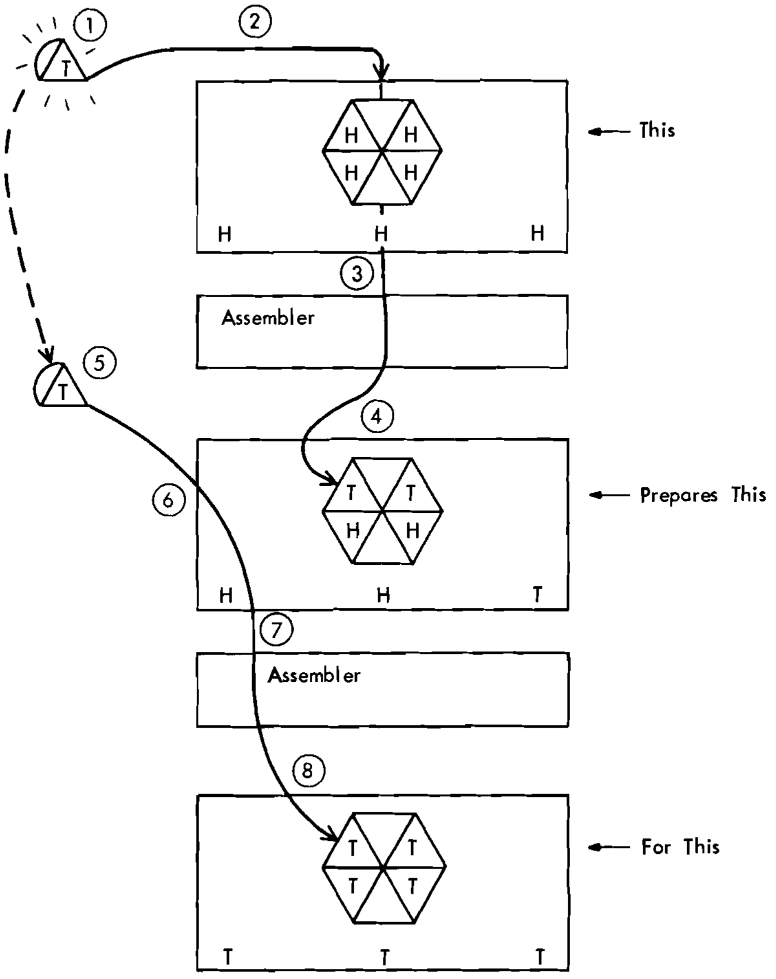


KEY: Dotted triangles indicate no manual changes (just "flush through").

Solid triangles indicate possible simple changes to program mechanics.

Highlighted triangles indicate reprogramming.

Fig. 6 The AED Bootstrap Process



STEP 1: Reprogram with target information.

STEP 2: Compile resulting programs on Host machine.

STEP 3: Assemble resulting programs on Host machine.

STEP 4: Load new Host-resident compiler with reprogrammed pieces.

STEP 5: Rework program mechanics for target machine.

STEP 6: Compile reworked programs on Host machine with the compiler generated in STEP 4.

STEP 7: Assemble resulting programs on Target machine.

STEP 8: Load target compiler with reworked programs.

Fig. 7 Example of Bootstrap Conversion Flow

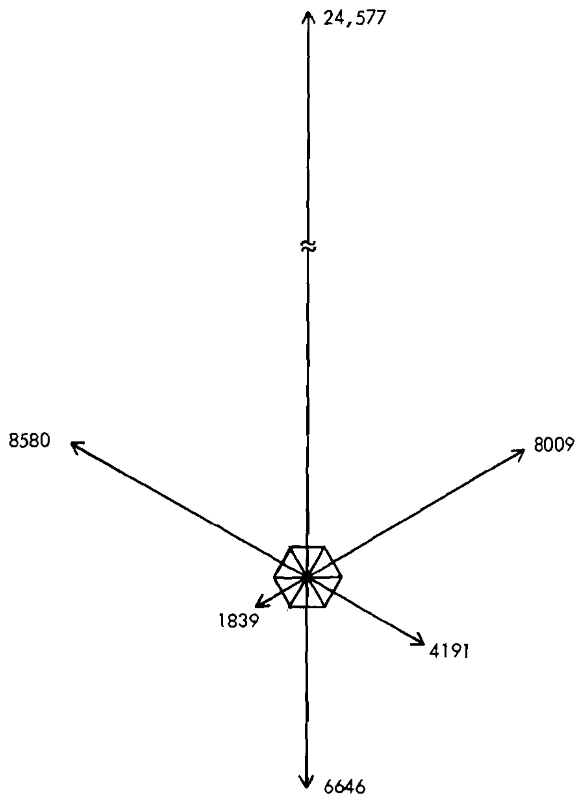


Fig. 8 Relative Sizes of 360 Hexagon Pieces (In 32-bit Words) for May 1968 Bootstrap

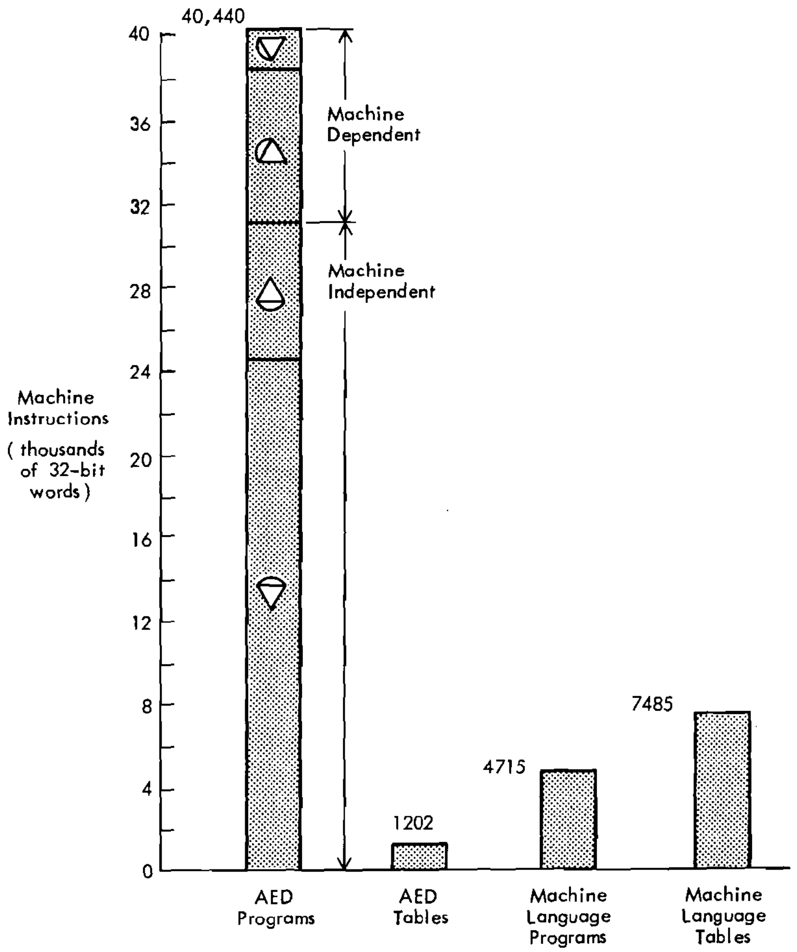


Fig. 9 Sizes of 360 Programs and Tables for May 1968 Bootstrap

Most of the "tables" are automatically generated by the RWORD and AEDJR Systems on the Host computer in the form of assembler macro calls. Therefore, the only step required to convert the tables for a new machine is to reprogram the macro definitions with Target machine information and process the macro calls through the existing Assembler on the Target computer.

The size of the machine-dependent AED-0 program portion will vary considerably depending upon the target computer. The 360 is by far the most complex yet used in this regard, so the 7378 program instructions required for this May, 1968 360 bootstrap should be taken as a maximum. The portion labeled "Machine Language Programs" in Fig. 9 have since been reduced by the introduction of the AED version of the ASEMBL output package. Also, some of the number-conversion routines and other of the Support machine language programs are most likely already available in the existing software for any given Target machine. Thus a considerable part of this section can usually be "stolen" with little reprogramming effort.

C. COOPERATION WITH OTHER GROUPS

The rather broad spectrum of cooperative work with many other groups, both within and without M.I.T., over the first eight years of the computer-aided design work has been recited in some detail in the previous final report, AFML-TR-68-206. During this final reporting period, as work concentrated more on the 360 and 1108 bootstraps, the M.I.T. association was primarily with Project MAC (except for the many M.I.T. users of AED on various M.I.T. computers). The number of visitors on the AED Cooperative Program was also somewhat less, but many additional companies and other outside organizations were working at their own locations with various-level system releases. These two main associations are summarized below; more details of the AED Cooperative Program are found in Chapter VII.

1. M.I.T. Project MAC

Project MAC at M.I.T. is an interdepartmental research activity in time-shared computing techniques, sponsored by the Armed Forces Research Project Agency. Since 1963 the MIT Computer-Aided Design Project has received very substantial support from Project MAC in the form of generous access to its powerful time-sharing facilities, and allocation of convenient laboratory and office space. In turn, all results of the Project have been made available to all users of the Project MAC facilities, (and later the duplicate facilities of the MIT Computation Center), and it is through this mechanism that the large number of other departments and projects have been able to make use of the Project results.

Over this same period, Project MAC has jointly sponsored (with the CAD Project) the work of the ESL Display Group. The work of this group during the reporting period, which has resulted in a number of joint publications, is described in Chapter VI.

2. The AED Cooperative Program

A very significant association of the Project with outside groups was the "AED Cooperative Program" which was in operation from March, 1964 to July, 1969. This program, which is described in more detail in Chapter VII, was a unique cooperative venture with industry whose primary purpose was to promote the dissemination and appreciation of the results of the CAD Project, while at the same time contributing toward their further advance. Experienced system programmers from industry joined our regular staff on a visiting basis for one year to learn about and contribute to the work of the Project. The contributions of the visiting staff members neatly balanced the educational load on our permanent staff so that not only was technical progress maintained, but ideas and skills which could only be transmitted by active participation were seeded in the most direct possible way into the vital activities of industry. Perhaps more than any other aspect of the Project, the AED Cooperative Program symbolized and made real the unique benefits which derive from a free and open intermingling of the talents and backgrounds of the academic and industrial community with the stimulation and support of government sponsorship.

A total of ten visitors from nine companies spent 98 man-months at M.I.T. during the reporting period. Participation over the entire five-year cooperative program was 31 visitors from 21 companies, and a grand total of 362 man-months, which represents a very substantial contribution to the AED effort by industry. Complete details of participation are presented in Chapter VII.

D. PROJECT TERMINATION

Mr. Douglas T. Ross served as Head of the ESL Computer Applications Group throughout the period of APT and AED development and the success of these programs is a result of his technical insight and leadership. With the phaseout of Air Force support for technical development in July, 1969, coincident with the release of Public AED, Mr. Ross and his key associates left M.I.T. to form a company (SofTech, Inc.) to continue AED-related work in the private sector. The period since July, 1969 has been devoted to completion and publication of AED documentation as described in Chapter III, to publication of several other reports as described herein, and to preparation and publication of this final report.

Distribution of the Version 3 AED/360 program tapes (on an at-cost basis for copying) is now being handled by SofTech, Inc., and they are also providing system maintenance and other services to AED users under a separate one-year Air Force contract.

CHAPTER II

FIRST PHASES OF BOOTSTRAPPING AED TO THE IBM 360 COMPUTER

All development work on AED at M.I.T. from 1963 on has been carried out on the MIT Compatible Time Sharing System, utilizing an IBM 7094 computer. This second-generation machine was already becoming obsolete in 1966, and it was recognized that AED would need to be made available on more-modern third-generation computers such as the IBM 360 and Univac 1108 if it was to be used in the aerospace industry. The 360 was chosen for initial efforts because of its wide usage, and plans for a bootstrap to the 360 were formulated in November, 1966, some six months prior to the start of the present reporting period (May, 1967). In order to put the progress under the present reporting period (May, 1967 to November, 1968) in proper perspective, we will first review this original schedule and the problems in meeting it (Sections A, B, and C). Sections D, E, and F then describe the completion of the initial bootstrap in January, 1968, and the successive Version 1 and Version 2 releases in May and September, 1968. The final phases leading to Version 3 release in July, 1969 are described in Chapter III.

A. THE NOVEMBER, 1966 360 BOOTSTRAP SCHEDULE

In November, 1966 (under the previous contract, AF-33(657)-10954), a tentative schedule had been established for bootstrapping the AED System to the IBM 360 computer. The schedule (with target dates) was set up with four phases as follows:

1. January, 1967 — Prepare the basic tools for modifying the AED-0 Second Pass to put out assembly language for various machines.

2. March, 1967 — Use those tools to compile AED-0 language into 360 assembly language.

3. July, 1967 — Modify the AED-0 source language programs for the entire system and bootstrap a basic working system onto the 360.
4. Fall, 1967 — Prepare a first system for others to experiment with.

Based on our experience in rewriting portions of the AED-0 System as it evolved in previous years, this appeared to be a tight but reasonable schedule.

The first phase of preparing the basic tools by January, 1967 was achieved nicely on schedule as was demonstrated at the Second AED Technical Meeting held that month at M.I.T.* Also the second phase of generating working 360 assembly language from the 7094 by the end of March, 1967 was essentially achieved on schedule in that all but a few AED language features were being compiled at that time.

Just prior to the start of the present reporting period in May, 1967, however, several events and problems occurred which had not been foreseen. Each event caused delay in the bootstrap process, so that the above schedule for phases 3 and 4 was not met. These events are described briefly below.

B. PROBLEMS IN MEETING PHASES 3 AND 4 OF THE NOVEMBER, 1966 SCHEDULE

1. Personnel Problems

Eight of the industrial visiting staff members of the AED Cooperative Program completed their periods of assignment at M.I.T. in March, 1967, and their departure required additional effort by the AED staff to complete checkout and documentation of their programming assignments. At the same time, the key member of the AED staff on compiler development withdrew from AED work temporarily

* Report AFML-TR-68-206, "Investigations in Computer-Aided Design for Numerically Controlled Production," Final Report for period 1 December, 1959 to 3 May, 1967, M.I.T. Electronic Systems Laboratory, May, 1968, pp. 151-160.

in order to devote full time to his M.I.T. doctoral program, and was not able to resume his AED efforts until September, 1967.

2. Computer System Problems

A set of technical problems arose that had to do with the 360 computer and its operating system. In spite of excellent cooperation from IBM and MIT Computation Center personnel, a job of this magnitude is beset with pitfalls. We had been using 360 computers at the IBM Cambridge Scientific Center and at the MIT Computation Center. Only IBM had a time-sharing system, and even there all AED work was funnelled through a single console. Also, the operating system rules changed several times as equipment was modified and updated, especially at the Computation Center, causing some lost time and much trauma.

We went through several schemes to transfer programs generated on the time-shared 7094 to these 360 systems, including the problem of converting from seven-track to nine-track tape and converting from BCD to EBCDIC code. Finally a fully automatic process was developed and implemented via a "runcom" command in time sharing, but still the shortage of disk tracks and limitations on the size of 360 runs forced files to be processed in small batches.

There was a minimum two-day turn-around time to modify, recompile, retransfer, reassemble, and rerun a file. Debugging took place in terms of machine patches which then were updated into the source decks, introducing another place for mechanical error. Operating on several computers also introduced further delays due to down time (on one computer or another)and scheduling conflicts, which occasionally caused an extra day or two to be slipped into schedule. The benefits of doing all work within a single powerful time-sharing system are painfully obvious when that system is not available. For a time, additional delays were incurred while we prepared tapes and tables for transmission to United Aircraft Corporation for the 1108 bootstrapping. Here again, several tries were needed to achieve the desired transmission in some cases, and since these activities used the same system resources and people, the 360 effort was unavoidably stretched out.

3. <u>360 Code-Generation Problems</u>

Another set of very time-consuming debugging runs resulted from the complexity of the OS/360 operating system. Many essential characteristics of a detailed nature were unknown and no source of the requisite information could be located. Therefore the preparation of the Input-Output Buffer Control Package (IOBCP) for the 360 took longer than anticipated.

Various characteristics of the 360 machine language and system organization make it a difficult computer for which to construct a sophisticated compiler. In particular, the use of base registers wrecks havoc with pointer mechanization and complicates subroutine calling mechanisms. From the beginning it was decided to make AED fully compatible with the standard Fortran IV calling sequences to yield a compatible system. Many of the characteristics of the 360 forced us to change our original plan of leaving the AED-0 Second Pass unchanged for generating 360 instead of 7094 machine code. Write-arounds to make the 360 behave like the 7094 would have been impossible or exorbitantly expensive. Therefore, several major and important parts of the Second Pass had to be rewritten from scratch, bringing in the additional debugging which we had sought to avoid. Other problems, such as the lack of a "movie table" in the loader, and the fact that the operating system allows only six entries per subroutine library added additional, though moderate, unscheduled burdens.

4. <u>File Editing Problems</u>

Until we started the conversion process, we had not realized the vast file-massaging work load involved in bootstrapping the AED System. We knew we would have to change the declaration portions of each program to accommodate the difference in word size between the 7094 and the 360, but had thought of this as a straightforward and quite routine matter. Because of the 360 machine code problems, however, it was found that we had to re-examine every single program to declare pointer variables as pointers rather than integers, as had been previously the case in AED-0. In some cases a value is used both as a pointer and as an integer, necessitating further re-work (and debugging). Also, in

AED-0 the use of labels had been allowed to violate Algol block structure, and the 360 implementation would not allow this. Therefore, once again, various programs required modification.

Finally, just the physical magnitude of the data processing load involved in all of these file manipulations forced us to make various major reorganizations of files in order to make use of the .INSERT feature to minimize the number of distinct copies of files and systems. Since we only learned the necessity for these requirements gradually from experience, many major source files were re-edited and rechecked two or three times.

We also added to our burden slightly by inserting remarks, comments and program documentation to the source files as they were reworked, so that they would be more understandable to future users and bootstrapping teams. This was a worthwhile investment of time, but caused further delay.

5. Compiler Problems

Another difficulty which had a profound effect that we did not anticipate at the beginning concerns the fact that the bootstrapping compiler itself was being debugged as we performed the bootstrapping. In our previous reworkings of the AED System, primarily during 1964, most of the early changes were made in the FAP assembly language, and since the FAP assembler was well debugged, only individual modules required change and reassembly as we modified the system. In the present case, however, whenever a compiler error was discovered, we were forced to recompile and reassemble all affected programs processed up to that point. Thus some of our earlier programs were processed completely three or four times, including all of the time delays of processing small numbers of files at one time, transferring from one machine to another, scheduling debugging time, etc., etc. Even though the source files did not always need further re-editing, this massive recompiling of programs greatly added to the tedium and delay in the process. This effect further intensified our interest in having the second pass of the compiler as machine independent as possible, since as long as the compiler itself is being debugged, the vicious circle is inherent in the bootstrapping concept and cannot be escaped.

6. New AED-0 Language and Subsystems

A final delaying factor was the unexpected need for additional software tools to accomplish the bootstraps. The 360 byte addressing (rather than 7094 word addressing) led to the need to distinguish between AED-0 pointer and integer data. Therefore, the AED-0 language itself was augmented slightly to allow for pointer declaration and a new form of equivalence to permit distinct pointer and integer symbols to refer to the same value. Improved code generation was needed, including efficient handling of transfer of control and testing of boolean expressions. An entire new subsystem was also created, called D.FEAT which permits automatic deletion of symbols from secondary declaration statements when primary declarations are deleted. This permits single large declaration files to be maintained for inserting in many source files, and a simple deletion (which may be done with the Feature Feature if desired) causes the coupled declaration statements to automatically be modified to conform. This not only allows complicated program changes to be kept in phase, but also yields shorter compiled programs.

C. REVISION OF 360 BOOTSTRAP SCHEDULE IN NOVEMBER, 1967

By the end of November, 1967, all of the above delaying factors were clearly understood, and a meaningful re-evaluation of the schedule was possible.

Looking back over the effort to that date, 200 major files, each containing many programs had been edited, compiled, and assembled into 360 machine code. Of those files, about 36 had undergone major rewrites, 12 of them for the new Second Pass features.

The 1966 AEDJR "like" mechanism yielded a much more reliable processing of the AED-0 language than the old system, and this was reflected in much more meaningful error comments to the user. AED-0 is a big, rich language. The AEDJR description of AED-0 involved 429 lines of vocabulary word definitions, and 1381 lines of "like" specifications, attribute declarations, etc. In addition almost 70 special execute programs had to be written to provide fine control over errors and elegant parsing. Actually, there were three separate parsings of subsets of the AED language used in the AED-1 processor to give the proper context control for symbol-table building, macro expansion, and code generation. It was anticipated that having all of this information in the convenient AEDJR form would make system maintenance and improvement much more flexible as the system evolved in the future.

As of November, 1967, approximately 320,000 bytes of 360 machine code had been generated, and we anticipated that the final system would be in the vicinity of 500,000 bytes once all the remaining packages and systems were bootstrapped. It was too early at that time to make any estimates on the minimum 360 configuration that would be required to run the complete system, and no sensible measures of operating efficiency were yet possible.

With these statistics in mind, a new schedule was set to complete the bootstrap of all programs to the 360 by the end of 1967. Since most subroutine packages and the AEDJR System had already been tested on the 360, the goal of having a working compiler before the end of January, 1968 was set, with a plateau release of "Version 1" shortly thereafter.

D. COMPLETION AND EVALUATION OF INITIAL BOOTSTRAP, (JANUARY, 1968)

The first successful AED-1 compilation on the 360 was performed as scheduled in January, 1968, followed by a testing period under the OS/360 environment in February. An overall evaluation of that initial version of AED-1 resulted in the conclusion that no release of AED-1 could be made, due to the size and efficiency of the compiler, which was prohibitive on a model 360 with 256K bytes of core (a common configuration for many of the potential industrial AED users).

However, by reorganizing the compiler passes slightly, recompiling the data structures with a more compact format, and certain other minor programming changes, a much better compiler seemed attainable with relatively little additional manpower and time investment. All modifications to the AED-1 logic stopped, pending the release of the revised system, scheduled for May, 1968.

As a first step in the process, due to the easy accessability of the 7094, the revisions were made and checked out using CTSS. This consumed the month of March. The 84 source files comprising the heart of the compiler were then bootstrapped to the 360, using the revised 360 data structure. With the maximum of about two boxes of assembler source input cards per run and 24 to 48 hours turn-around on the 360, along with errors in the new data structure definition which caused several recompilations, the total bootstrapping process took the month of April.

The month of May, 1968 was spent catching obscure bugs in AED-1, such as errors in output format when a single ".C." character string exceeded a single assembler card in length, faulty compilation of the phrase COMP (LOC ATOM), etc. This shakedown process proceeded as rapidly as possible, with a minimum of 48 hours between discovery of a bug, bootstrap compiling the correction, 360 assembly of the resultant machine language program, and redoing the link-edit of the 360 compiler to incorporate the correction on the 360.

E. VERSION 1 RELEASE OF AED/360 (MAY, 1968)

The May, 1968 schedule was met, and a field-trial, special release of a slightly restricted version of the total AED System was made for those users who wished an early version for experimentation purposes. This was known as Version 1 AED/360. A copy of this system was sent to those users who sent a written request, along with a reel of magnetic computer tape on which to copy the system.

The May, 1968, release contained the following items:

1) The AED-1 Compiler, with the following restrictions:
   a. No PRESET language
   b. No BLEBR language
   c. No PRINT, READ, and FORMAT statements
   d. No Macro Pass (and therefore no PRALG option)

2) The AEDJR System (complete)

3) The following packages:
   a. Free Storage (complete, zone-structured version)
   b. Input-Output Buffer Control Package (IOBCP)
   c. Generalized Output (ASEMBL)
   d. Generalized Input (RWORD)
      (RWORD3 only. RWORD1 and RWORD2 not yet available)
   e. Other small user packages
      (ISARG, DOIT, NUMTOC, etc.)

The missing pieces (PRESET compiler language, compiler Macro Pass, RWORD1 and 2, etc.) were still being debugged, and a full, complete system was scheduled to be distributed as a Production Release in the Fall of 1968.

At this point, sufficient experience had been gained using AED-1 and AEDJR on the 360 to permit publication of some figures concerning execution time and memory requirements.

1. Size Considerations (Version 1)

Generally, the AED System Version 1 would easily fit on a 360 model 40 or larger, with a minimum of 256K bytes of core memory and direct-data disk facilities. Specifically, the compiler required approximately 30,000 decimal words and AEDJR about 22,000 decimal words of core memory, not counting the 360 Operating System. Comparing these sizes to the equivalent 7094 programs, there was an increase of between 25 to 35 percent in 360 program size.

This increase was due to several things, but the two major reasons were (1) size of data tables and (2) size of the instruction sequences required to enter and leave procedures. To elaborate on these, the 360's 32-bit word size clearly cannot contain as much packed data as the 7094's 36-bit word. In addition "pointers" require 24 bits (because of byte-addressing) on the 360, whereas they only require 15 bits on the 7094. This restriction permits only one pointer per word on the 360, whereas the 7094 permits two per word. Since both AED-1 and AEDJR make heavy use of pointers packed into large data structures, they pay the resulting space penalty.

The inefficiency of the call and return mechanism necessary to enter and leave procedure bodies was a consequence of the decision to be compatible with standard IBM Fortran conventions. That is, every time an AED procedure is entered or left, certain machine conditions must be saved and restored in a particular way to maintain compatibility with other compilers such as FORTRAN. This cost heavily in both space and time, but had to be done for compatibility. To remedy this inefficiency, the AED staff devised a new instruction sequence which cut both the time and space of the enter/leave sequences in half, while still maintaining compatibility. To incorporate this change into the AED-1 files would, of course, require reassembling all decks with the new sequence. This was planned for the Version 2 release, and it was estimated that a saving of 8,000 bytes of memory would result, plus significant time savings.

2. Execution Time (Version 1)

When considering efficiency of execution time, a distinction had to be made between the cost of I/O operations, operating system overhead costs, the cost of calling procedures versus "in-line" coding, the cost of logical-type statements versus arithmetic operations, etc. It was not feasible for the AED Project to expend the effort at that time to perform such an in-depth analysis. However, sufficient experience had been gained to touch on most of these areas and to at least give a good, over-all picture of the execution-time efficiency problem. The AEDJR System was used for the majority of the study, since it had been in operation on the 360 for a much longer period of time than AED-1.

Several different runs were performed with the same identical AEDJR deck on the MIT360 and 7094/CTSS Systems. The deck used contained 44 VDEF cards, a "FRESH RUN*" card, and 18 cards in the language defined by the VDEF statements. The deck was run with all possible I/O active (LOUD RUN, SHOW ALLALL, STATE, and SIM LONG AEDJR options), and then with no printed output requested in an attempt to see the cost of the AEDJR debugging printouts. The same deck was run on the 7094 and 360. The 7094 runs were performed several times, and the average run time was calculated, since these numbers vary somewhat due to machine activity. The "swap time" print given in the CTSS "ready message" was considered the system overhead.

On the 360, a special statistics job was prepared which read the computer clock immediately before and after the AEDJR System was called and printed the difference between the two readings. Due to variations in clock readings, these runs were performed several times and an average time calculated. By subtracting this time from the total time charged at the end of the job, the 360 overhead charge was determined. Lastly, the 360 AEDJR overlay system was run to determine the cost of running the overlay versus non-overlay versions. There was no detectable difference in the timing of the two versions for the small test case run. The results of all these runs are shown in Table I.

3. Evaluation (Version 1)

There are several aspects of the Version 1 statistics given above which should be pointed out to avoid drawing wrong conclusions. First, the CTSS console print is very expensive in swap time, execution time, and programmer time required to sit and wait for a lengthy print to be typed on a CTSS console. Therefore, it is doubtful that the run entitled "7094 with Console Print" represents a common situation, but it is presented for completeness.

Another pertinent fact is that the charging technique employed for the MIT 360/65 does not bill the user for read-in or print-out time, since the ASP system delegates this task to the 360 model 40, which is considered "off-line". Also, the charges quoted were M.I.T. rates which are considerably less than commercial computer time charges.

Although we were pleased with the initial efficiency of the 360 AED Version 1 programs, it seemed likely that the future would bring dramatic reduction in both size and execution times as the system was "polished". On the other hand, no improvements in 7094 AEDJR time were foreseeable. For example, the 360 Version 1 subroutine entry and exit techniques employed were to be changed, cutting the enter/leave execution time in half. Because of the extensive use of procedure calls for modularity in AEDJR, a noticeable increase in efficiency was foreseen.

Table I

Comparison of Run Time and Cost for
Version 1 AEDJR on the 7094 and the 360

7094 WITH CONSOLE PRINT

	AEDJR	SYSTEM OVERHEAD	TOTAL
TIME (SEC)	9.85	11.75	21.60
COST (PRIME)	$0.82	$ 0.98	$ 1.80
COST (LOW PRIORITY)	$0.55	$ 0.65	$ 1.20

7094 WITHOUT CONSOLE PRINT

	AEDJR	SYSTEM OVERHEAD	TOTAL
TIME (SEC)	3.45	$ 3.80	7.25
COST (PRIME)	$0.29	$ 0.31	$ 0.60
COST (LOW PRIORITY)	$0.19	$ 0.21	$ 0.40

360 WITH LINE PRINT

	AEDJR	SYSTEM OVERHEAD	TOTAL
TIME (SEC)	3.80	8.80	12.60
COST (HIGH PRIORITY)	$0.32	$ 0.73	$ 1.05
COST (LOW PRIORITY)	$0.21	$ 0.49	$ 0.70

360 WITHOUT LINE PRINT

	AEDJR	SYSTEM OVERHEAD	TOTAL
TIME (SEC)	2.05	6.95	9.00
COST (HIGH PRIORITY)	$0.17	$ 0.58	$ 0.75
COST (LOW PRIORITY)	$0.11	$ 0.39	$ 0.50

"System overhead" also deserves more careful consideration. The CTSS charges given above did not include the user access charge, console rental, user file directory charge, or disk storage charge, whereas the 360 charges were really total cost. Therefore, some fixed percentage should really have been added to the 7094 cost figures, although this amount was unclear.

Another aspect of system overhead should be mentioned to place it in perspective. Namely, the 360 overhead amount is essentially constant no matter how large the AEDJR job, since it costs only so much to bring in AEDJR from disk to core and process the job control cards to start and stop the job. However, CTSS overhead (swap time) varies dramatically depending upon number of active users on the system and how large a core image is created by the AEDJR run, plus several other minor variations.

Finally, programmer time and real time required to debug programs are not considered. Whereas it is possible to process at most two or perhaps three runs of a single job in one day with the 360 batch-processing technique, the 7094/CTSS system affords an essentially unlimited number of runs per day. This aspect cuts the real-time program debugging time dramatically on CTSS, but also probably increases the number of runs, since less desk-checking results from the easy machine access afforded by CTSS.

F. VERSION 2 RELEASE (SEPTEMBER, 1968)

After the release of the Version 1 plateau in May, 1968, the AED Project compiler efforts concentrated on the following areas:

1) Bootstrap the remaining pieces not included in the Version 1 plateau, heading toward a Version 2 plateau.

2) Receive and act upon suggestions, error reports, etc. from the Version 1 field trial users.

3) Distribute additional copies of the Version 1 system upon request.

An organized checkout was begun using the special test cases formerly used to check out AED-0 on the 7094. Each of these eleven test cases exercises one or two language features, so that once all of these tests were successfully compiled and executed, the majority of the basic as well as the more obscure statement forms would be checked.

Work on Version 2 began with the PRESET and BLEBR features. An investigation also began on the feasibility of incorporating the Octal Stream Package.

The Alarm Package effort was revived, along with a set of more detailed AED-1 error messages to be incorporated as an additional-compiler overlay which is called in if any alarms have occurred during compilation. This alarm reporting was scheduled to be incorporated as soon as possible, since the Version 1 alarms were designed for debugging and not for general users. Several minor changes in the Alarm Package seemed desirable, and the changes were made.

The Macro Pass underwent some revisions in preparation for hooking it up to the rest of the compiler. Specifically, the following steps were scheduled:

1) Transferring of some "executes" from Phase I to the Macro Phase.

2) "Marking" of the macro vocabulary tables.

3) Shifting to the AED-1 data structure.

4) Change of control from the general-purpose AEDJR System to a hand-tailored production version.

5) Processing of partly processed .INSERT files.

6) Bootstrapping to the 360.

Step 1 was begun during May, 1968. The various schemes for more efficient procedure enter/leave macros were re-examined and two versions were coded. The efficient version permits AED procedure to call on non-AED procedures only, while the less efficient version also permits non-AED procedures to call upon AED procedures. Checkout of these schemes began soon after release of Version 1.

During the month of June, 1968, five more copies of Version 1 were sent on request, making a total of eight companies having the Version 1 system. No communications had been received regarding problems encountered in getting the system to operate properly, so it was assumed that the distribution techniques were working well. Meanwhile, shakedown continued at M.I.T. Several minor bugs were uncovered and corrected. The archive of special test cases for the AED-0 Language features was helpful in this regard. Several of the tests required the PRESET facility, which was not yet available until Version 2.

During June, work on Version 2 continued, and all programs required to handle the PRESET facility were written. A new version of AED-1 was generated on the 7094 to include new, more efficient linkage conventions at the entry and exit of subroutines. This version also had an improvement in the code generated for one-bit components and storage-to-storage moving operations. These changes were incorporated into the 360 AED-1.

During July, 1968, the initial version of the programs to process the PRESET language were debugged, but it was decided to rework the programs to move some of the Phase III programs to Phase II.

In August, 1968, Version 2 of AED-1, containing the PRESET feature was working on both the 7094 and 360. Some additional shakedown was foreseen before release of the new system to participants. Files were reorganized in preparation for the release. The modifications to add the full alarm print to AED-1 were also completed and working on the 7094. Bootstrapping to add this feature to the 360 began shortly thereafter. Macro Pass debugging continued. The first "real-life" user of the system revealed some problems, and these were corrected.

During September, tests of Version 2 of the 360/AED-1 Compiler, AEDJR, and AED subroutine packages were completed, and copies of Version 2 were distributed to those companies having magnetic computer tapes at M.I.T. of the new release. By the end of Contract F33615-67-C-1530 on November 30, 1968, thirteen copies of Version 2 had been released, including a copy sent to Bell Helicopter (Fort Worth) during November. Discussions with these companies during November were very helpful in locating compiler bugs and in discussing the process of setting up problem-oriented systems with AEDJR.

Meanwhile, work had continued at M.I.T. on the remaining pieces of AED-1, working toward a Version 3 which would include the Macro Preprocessor. This work, which was conducted under the follow-on Contract F33615-69-C-1341, is discussed in Chapter III.

G. AED QUESTIONNAIRES

Subsequent to the Version 2 release, it was felt that "outside" recipients of various field-trial versions over the preceding three years had perhaps had enough experience with them to provide some valuable feedback concerning the comparison of AED with other available programming systems. Thus, with the approval of the Project Monitor, a questionnaire, shown in Appendix A, was prepared and distributed to all recipients of AED systems and documentation. These questionnaires were mailed in late May, 1969, with a requested return date of June 27.

Fifty-four completed questionnaires were received. The results indicated that ten organizations had actually used AED, that some 60 engineers and programmers had learned AED, and that AED had been used in 17 different programming projects. Usage of various field-trial versions was as follows:

Computer	Usage
7094	7
1108 Exec2	4
360 OS	6
360 CP/CMS	4

One question asked for free-form comments as to the strongest and weakest features of AED. The following were the most frequently cited attributes or weaknesses (with number of mentions):

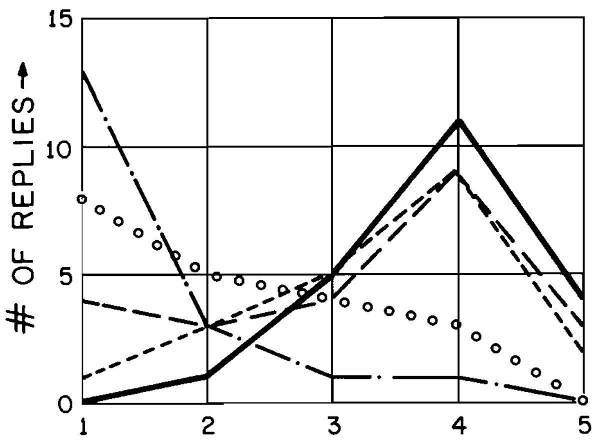
Strongest		Weakest	
Generality	13	Documentation	18
Data Structuring	12	I/O Handling	5
Flexibility	8	Speed	4
Language	7	Size	3
Free Storage	7		
System Building	6		
Program Structure	3		
AEDJR	3		
RWORD	2		

From the above, it was clear that lack of adequate documentation was a major issue, but work on new documentation was already well under way, as will be described in Section C of the next chapter.

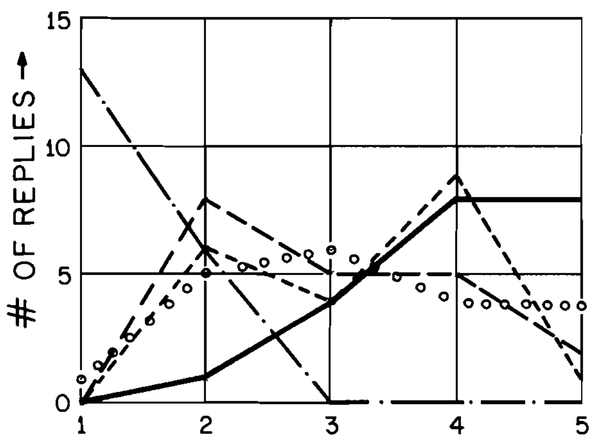
Another question asked recipients to make comparative ratings of AED with three other high-level languages, and with assembly (machine) language, for use in five major areas of programming: research investigations, general programming, system building, graphics, and data base management (there was also an "other category, but so few replies were received that it has been omitted from analysis). The results, graphed in Fig. 10, indicate that the potential of AED was most clearly recognized in the areas of system building, graphics, and data base management, although Fortran was preferred for general programming.

Another way of looking at the results is given in the bar chart of Fig. 11. Here the two weakest (1 and 2) ratings for all categories have been summed and averaged and shown in white. Similarly, the shaded bars show overall averages for the two strongest ratings (4 and 5).

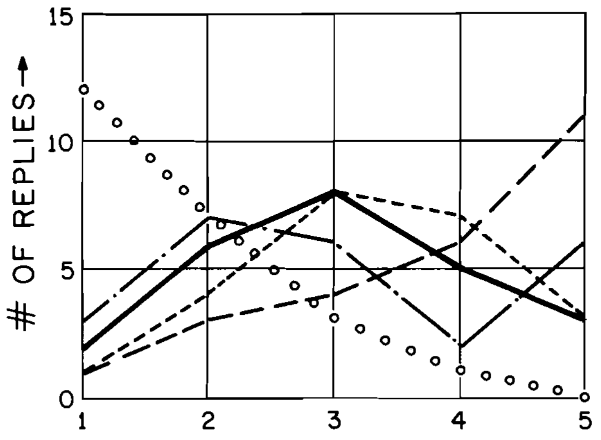
It is realized that these comments and opinions are based on a fairly small sample. However, they do indicate that those who have had the opportunity to become acquainted with AED recognize its present capabilities and potential for the future.



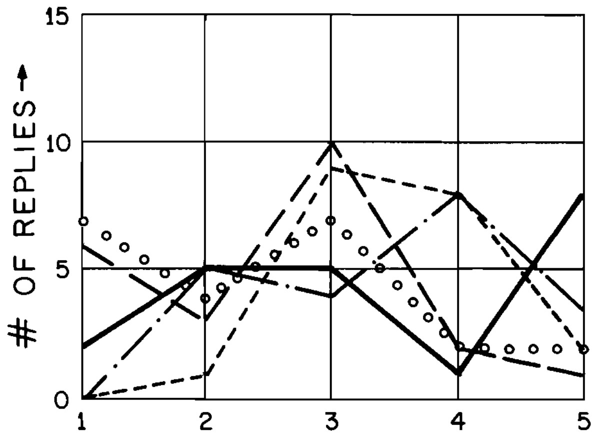
(a) Research Investigations



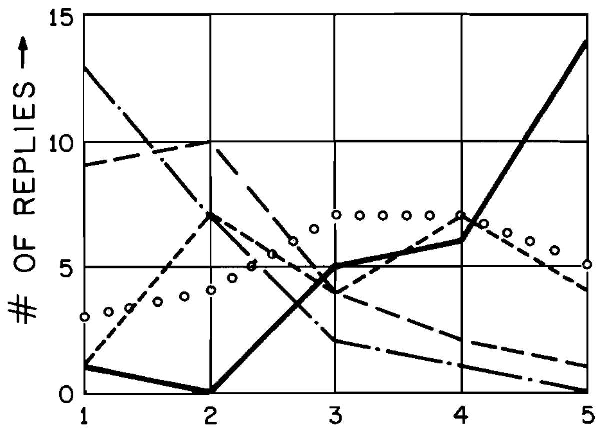
(d) Computer Graphics



(b) General Programming



(e) Data Base and Management Systems



(c) System Building

Notes:

━━━ = AED
— — — = FORTRAN
- - - - - = PL/1
—·— = COBOL
o o o o = ASSEMBLY

1 = VERY LOW RATING
5 = VERY HIGH RATING

Fig. 10 Summary of 54 Questionnaire Returns on Comparison of AED with other Languages for Various Programming Jobs

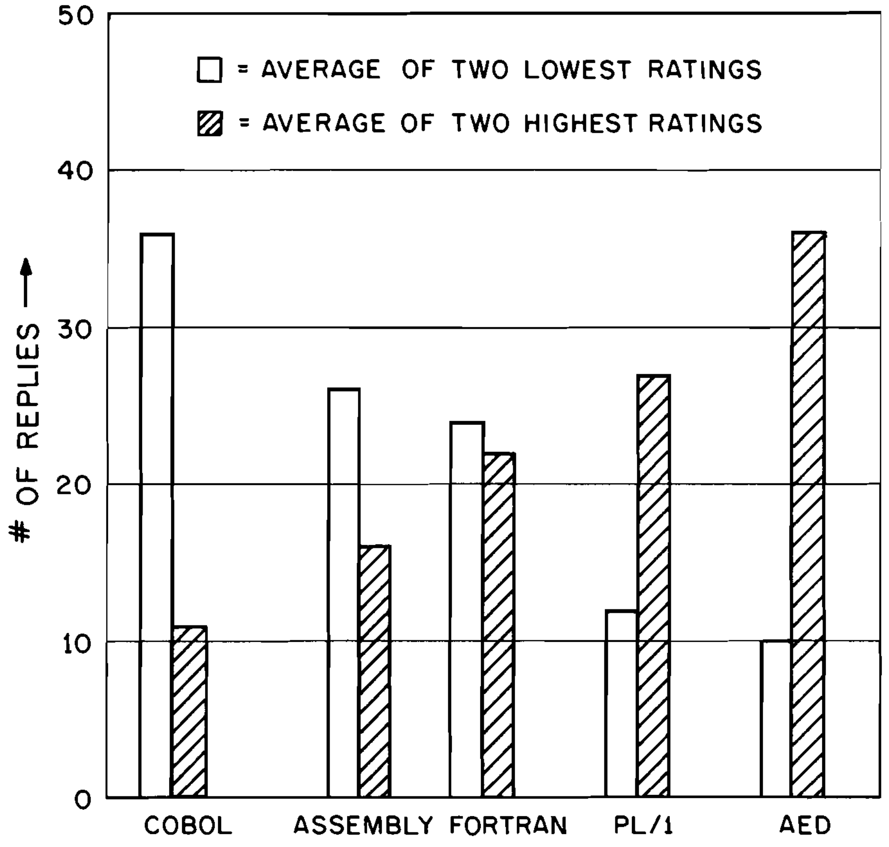


Fig. 11 Composite Ratings for all Programming Areas (54 Questionnaire Returns)

CHAPTER III

THE FINAL (VERSION 3) AED/360 RELEASE

The Version 1 and Version 2 AED/360 releases described in the previous chapter were considered to be interim field-trial systems.

Work during the final phase of the M.I.T. Computer-Aided Design Project in the seven-month period from 1 December 1968 to 31 July 1969 under Contract F-33615-69-C-1341 was concentrated on achieving and properly documenting a fully releasable AED system for IBM 360-series computers. This final MIT system, which became known as Version 3(or "Public") AED, was formally announced at the Third AED Technical Meeting held at M.I.T. on July 15, 1969.

This period was one of frantic activity in incorporating additional features available in the 7094 CTSS version of AED, making design improvements to improve efficiency, and documenting features of the language and systems which were still subject to change right up to the end. This activity can best be described by presenting the significant changes incorporated into Version 3, a detailed description of the contents and format of the Version 3 release tapes, and a description of the documentation that was prepared.

A. NEW FEATURES OF AED VERSION 3

During the final stages of work leading up to the Version 2 release in September, 1968, a certain amount of work had been done in preparation for the later Version 3. The Macro Preprocessor had been checked out on the 7094, but before compiling the programs for other computer, it was decided to reorganize the programs to increase the efficiency of macro call processing. The revised programs expanded macro calls using a single pass over the macro body instead of one pass to substitute arguments and perform other preliminary processing, and a second pass to finish the macro expansion.

The Macro Pass checkout was completed on the 7094, and all of the source programs were compiled for the 360. Debugging of the 360 Macro Pass was about to begin. The "external data" feature was also added to AED-1 to allow the user to distinguish between external <u>data</u> and external <u>programs</u> instead of the trickery formerly required.

This new facility was added to the 7094 version of AED-1 during November, 1968, and was added to the 360 version shortly thereafter.

Work on the compiler alarms had also continued. The review of all of the compiler alarm messages was concluded, and the new messages were typed into the alarm files in the 7094 CTSS System. Since additional alarm data was needed for the new, more specific messages, several alarm procedure calls had to be changed. At the same time, a "terminating phase" number was added to each alarm call, and the alarm processing program was changed to examine this data and terminate the compilation at the end of the phase.

The most significant change introduced in the Version 3 compiler was the redesign of the procedure linkage scheme. The Version 3 linkage scheme reduced both the procedure call time and procedure body storage overheads. For example, the time overhead per procedure call was reduced by a minimum of 30% for non-recursive procedures and by a factor of 10 for recursive procedures. The fixed storage overhead per procedures was reduced by a factor of two.

The principal design change which permitted the improved performance was the assignment of 360 general register 12 to contain permanently the address of a control section where all utility routines are located. In order to obtain the maximum benefit from this convention the contents of register 12 must be guaranteed at the entry to an AED procedure. This is, of course, not possible when a non-AED procedure calls an AED procedure. Thus two modes of compilation were implemented: the non-compatible and the compatible mode. In the non-compatible mode it is assumed that register 12 is properly set upon entry to a procedure. In the compatible mode no assumptions are made as to the contents of register 12. The standard compilation mode is the non-compatible mode. The compatible mode is forced by specifying the option FORT.

Although it is possible to mix procedures with the Version 2 and Version 3 linkages, difficulties will be encountered during abnormal returns which encounter both types of procedures on the stack. Also, access to optional arguments of Version 2 procedures with the Version 3 ISARG package will not work properly. Because of these difficulties and because all future maintenance will only cover Version 3 it is strongly recommended that all programs be recompiled as soon as possible with the Version 3 compiler. Programs written in assembly language using the ENTER, LEAVE and GETSAV macros should require minimal changes as long as the spirit of the macros was not violated by excessive knowledge of their expansion.

The Version 3 system corrects certain omissions of previous AED systems as well as introducing some features not previously announced.

1. Additions to the Compiler

a. The stack manipulation language (BLEBRS) was implemented.

b. The POWER (**) operator was implemented.

2. Additions to the Language

An EXTERNAL data facility was added to the language. The new reserved word EXTERNAL is used to declare that an identifier of type REAL, INTEGER, BOOLEAN or POINTER (scalar or array) is an external data item.

An external identifier must be defined in precisely one compilation and referenced from all other compilations in which it is declared EXTERNAL. A definition is distinguished from a reference by requiring that the identifier be PRESET in that compilation which defines it. Since the actual linking of references to definition is done by name through the loader (linkage editor), multiple definitions are detected at loading time by that program.

The syntax of the external declaration is:

```
EXTERNAL identifier [, identifier]* $,
```

Example:

```
BEGIN
POINTER A, B $,
INTEGER ARRAY C(10) $,
EXTERNAL A, B, C  $,
PRESET A = B $,
    .
    .
    .
END FINI
```

In this example, A is an external pointer defined in this compilation, B is an external pointer and C is an external integer array both of which are only referenced in this compilation and thus must be defined elsewhere for a successful execution.

3. Implementation Changes

a. The procedure linkage mechanism was changed as follows:

General Register 12 has been assigned to contain permanently the address of control section $AEDLNKG. The $AEDLNKG control section contains those programs which effect the call and return mechanism as well as several common routines (i.e., mode conversion, EXIT, ABEXIT).

Register 12 must never be destroyed.

Register 12 is loaded with the address of $AEDLNKG in one of two ways:

1. At entry to an AED external procedure whose name is MAIN.

2. At entry to any AED procedure compiled using the FORT option.
The FORT option allows AED procedures to be called from FORTRAN compiled programs. An AED procedure can always call a FORTRAN subprogram since the FORTRAN call-return mechanism saves and restores all necessary registers. However, a FORTRAN subprogram may only call AED procedures compiled with the FORT option since the contents of Register 12 while in a FORTRAN subprogram is unpredictable.

b. The implementation of label and procedures passed as arguments is consistent with the value of the LOC operator.

The address passed in an argument list for a label or procedure Q agrees with the value of a pointer variable P set by the statement P = LOC Q.

This implementation change affects users of the DOIT package and of packages having exchange functions for procedures arguments.

1. Use of DOIT for abnormal return.

   One application of the DOIT package is to set up a label as a global error return point by means of a pointer variable and use this pointer as the argument of DOIT whenever one wants to "re-start" the program. This usage of DOIT guarantees that the dynamic storage in the stack of active procedures is properly returned.

   To set up the pointer variable one writes:

```
        LABL.POINTER = LOC  ERROR.RETURN $,
ERROR.RETURN $ ...
```

   Then, to effect a transfer to ERROR.RETURN from anywhere one writes:

```
                DOIT (LABL.POINTER) $,
```

   This mechanism works as long as the invocation of the procedure which set up LABL.POINTER is still active, i.e., has not returned. In previous versions of the AED System, this mechanism was made to work by writing:

```
                DOIT (WHOLE (LABL.POINTER)) $,
```

   where WHOLE is an unpacked pointer component with \$=\$ value zero.

2. Use of exchange functions with procedure arguments.

   Packages allowing user supplied procedures to perform interface functions occasionally use a method of specifying such functions called the exchange procedure. In an exchange procedure the user supplies as an argument the new value of a parameter and receives upon return the old value of that parameter. When the value that is being exchanged is a procedure, the only means of returning the old value in AED-0 is by means of the catch-all type pointer. Thus, what the exchange procedure actually returns in this instance is the LOC of the old value. Consequently if one wishes to restore the old value of the parameter, this extra level of indirection must be removed by means of an unpacked component with \$=\$ value of zero.

The following example illustrates the technique:

Procedure SETOUT is used to set the output procedure in the ASEMBL package. The form of the call to SETOUT is :

```
        old.proc  =  SETOUT(new.proc)  $,
```

where

```
        new.proc     is a PROCEDURE
and     old.proc     is a POINTER
```

In order to establish the old procedure WHERE LOC is set in old.proc one must issue the following call:

```
        SETOUT (WHOLE(old.proc))  $,
```

where

WHOLE is an unpackaged pointer component with

$=$ 0.

c. In the SYSIN file produced by the compiler, double-indexed instructions using only one of the two possible registers now always use the base register field. Previous versions used the index register field. This change should save around 10% of the total CPU time.

d. Argument lists of procedure calls are assembled at the end, after DATA, instead of in-line. This saves the time and storage of a NOPR in most calls.

e. Statements of the form:

```
    A  =  B    $,
    A(B)  =  C(D)  $,
```

compile, when nesting conditions permit it, into more efficient code using move instructions, NI instruction, or 0I instructions depending on the types and value of both sides of the "=".

f. The first word of a CSECT contains a 0 in the first byte and the address of the static data area (DATA) in the last 3 bytes.

4. New System Features

a. MACRO Preprocessor

The AED MACRO pre-processor is available in Version 3 for the first time, and is available as a stand-alone processor independent of the compiler.

b. Debugging Facility

The Version 3 system provides a library containing a debug version of the $AEDLNKG control section and a package for selective tracing and other debugging functions.

The debug $AEDLNKG provides a basic facility of calling a tracing procedure upon entry, exit and abnormal exit of any procedure. The names of the procedures for which tracing is desired and the LOC of the corresponding procedures to call are obtained from a user supplied external array (.AEDBG).

The basic facility provided by the debug $AEDLNKG is exploited by the TRACE package to allow the user to:

1. TRACE any procedure by name
2. INTERRUPT execution at entry and/or exit of any user procedure
3. Examine and modify memory while 'interrupted'.
4. Change the trace and/or stop information while 'interrupted'
5. Specify any user tracing procedure while 'interrupted'

The interrupt facility can be effectively used only on a time-sharing environment.

c. A new free storage package incorporating most of the features of the large free storage system but which provides higher performance and a cleaner operating system interface.

d. Numerous improvements in the labrary at the operational level.

B. CONTENTS OF AED VERSION 3 RELEASE TAPES

The following describes the format and contents of the tapes included in the Version 3 AED release. It also describes how to use the tapes to activate AED at a new installation, and gives sizes of various setups under both batch (OS/360) and time sharing (CP/CMS).

Version 3 of AED-1 for the IBM System 360 Computer is contained on four reels of magnetic computer tape. The four tapes are divided into the following logical categories, with one tape for OS/360 users, two tapes for CP/CMS users, and one tape for both OS and CP/CMS users:

Tape No.	Contents	Operating System
1	Source Programs	OS/360 and CP/CMS
2	Partitioned Data Sets for Running AED	OS/360
3	Modules and Libraries for Running AED	CP/CMS
4	Text Files	CP/CMS

The remainder of this section describes the tape contents in more detail, and describes how to use the tapes to get AED in operation at a 360 installation.

1. Tape 1: Source Programs

Tape 1 contains the 540 source programs for the AED-1 Compiler, the AEDJR System, and the AED Subroutine Libraries. The tape also includes two simple test cases, one for the compiler and one for AEDJR. They are included to give confidence that no malfunctions (such as bad tapes) have occurred in the copy process.

The tape has a standard tape label with identification AED001 and is blocked at 800 bytes (10 cards) per record. The master tape is recorded at 800 B.P.I. density in 9-track form, but copies may be requested at other densities and in 7-track format, if desired. The total tape contains approximately 64,000 cards of EBCDIC source data written primarily in the AED-0 Language, with a few 360 Assembly Language, AEDJR, and RWORD System input-language programs. The programs are grouped into 11 files, as follows:

File No.	No. of Programs	Tape Records	DSNAME	Contents
1	104	2078	AEDSYS01	AED-1 Compiler
2	88	565	AEDSYS02	AED Library (AED-0 Source Programs)
3	92	707	AEDSYS03	AED Library (360 Assembly Language Programs)
4	17	47	AEDSYS04	Assembler Macro Library
5	35	217	AEDSYS05	"Fast Free" Free Storage Package
6	44	420	AEDSYS06	.INSERT Files (declarations)
7	67	802	AEDSYS07	AEDJR System Source
8	70	1059	AEDSYS08	AED-1 Macro Pass Source
9	10	253	AEDSYS09	TRACE Debugging Package
10	13	222	AEDSYS10	SHOWIT AEDJR Example
11	2	110	AEDSYS11	Simple Test Cases

The tape uses the standard IBM distribution method for source programs: input to the 360 IEBUPDAT program, a basic OS/360 utility for creating and editing source program libraries. The tape contents are designed to create a single 360 Partitioned Data Set source library for each of the ten files listed above. To create such a library, the user selects the desired portion of the AED source distribution and then runs a 360 IEBUPDAT job, specifying the corresponding tape file as input, as well as the device, name, size, and format for the library he wishes to create. Once the source library is created, members may be edited, compiled, printed, punched into cards, or whatever is desired, using the basic OS/360 utilities. Details of how to run such jobs are left to the programmer at each installation, since they are common, well-documented OS/360 jobs.

To make proper use of the tape, it is necessary for the programmer to know the precise sequence and format of the tape contents, and the following paragraphs discuss these details. Each tape file consists of a sequence of IEBUPDAT "ADD cards" and source programs, where a single ADD card precedes each program, and has the following format:

Columns 1, 2	Columns 10-12	Columns 16-72	Columns 73-80
./	ADD	prog.name, 00, 0, 0	PPPNNNN3

Prog. name is the name of the program that follows the ADD card, PPP is the program's identification number, NNNN is the card number within the program, and 3 is used to indicate "Version 3". The prog. names are ordered alphabetically, as required by IEBUPDAT. There are no JCL cards on the tape, and the very first card is an ADD card for the first program. The sequence field for each tape file is entirely numeric, and is sequential, with the smallest number at the beginning and the largest number at the end of the file. PPP program numbers begin with program 001 and increase by 1 for each new program. NNNN card numbers increase by 4 for each successive card, leaving room for 3 insertions between adjacent cards before card replacement is necessary. PPP is reset to 001 at the beginning of each of the 10 files.

Because of the large bulk of information, it is desirable to keep only one single source tape for both OS and CP/CMS operating systems. The vast majority of programs are operating-system independent, and any programs which are system dependent are identified by the letters "CM" (for CP/CMS) or "OS" (for OS/360) at the end of the prog.name parameter of the ADD card. For example, tape 1 contains programs MCZNCM and MCZNOS (programs 63 and 64, file 8) for the CMS and OS versions of macro pass source program MCZN. Future updates and corrections to the tape will be made using the established numbering scheme, so it is imperative that the sequence of data not be altered by the tape recipients.

To be able to use the same tape on CP/CMS, a special program named OSTOCMS is distributed as file 4 of tape 3 (described below). This program may be used to read an IEBUPDAT tape in the format just described, copying selected programs from selected files into the user's CMS file directory. See the more complete description of OSTOCMS, below.

Before proceeding to discussion of tape 2, a warning is in order. Most programs may be compiled or assembled directly from the source libraries generated from tape 1, without ever editing the programs or punching them onto cards. However, certain precautions are in order. Since AED-0, assembly language, AEDJR, and RWORD source programs are mixed in all tape files except for files 2 and 3, wholesale 360 jobs such as "compile all programs in AEDJR" should be careful to distinguish between the various source languages. Also, most AED-0 programs contain one or more .INSERT statements, and require these inserted files to be correctly identified in the JCL for each compilation. Lastly, a few programs contain .FEAT. cards for the AED "Features Feature", and these programs are not directly compilable as distributed. In all cases, unfeaturized versions for some of the more common setups are also included on the tape, and immediately follow the featurized version on the tape. Comment cards at the beginning of these files describe the features included, and the prog.name mnemonics consist of the name of the featurized version followed by the version number. For example, the first three programs in the AEDJR Library are AJAA, AJAA1, and AJAA2 for the featurized AJAA program, plus two unfeaturized versions.

2. Tape 2: OS/360 Partitioned Data Sets

Tape 2 contains the OS modules and libraries required to run AED on the 360 under OS. All modules are designed to work under PCP, MFT, and MVT. A DOS version has not been considered, and users are warned not to attempt to use AED under DOS.

The tape in the form of an "unloaded" disk pack (model 2311) uses the 360 IEHMOVE utility program. That is, the contents of the disk pack being used to run AED at MIT were written onto tape by IEHMOVE COPY PDS commands, so that the reverse process of COPY PDS commands from tape 2 to the user's disk will result in an identical copy of the MIT disk contents. Actually, the IEHMOVE job can be instructed to copy the files onto a wide variety of memory devices as dictated by the JCL statements, but the remainder of this description uses 2311 disk pack statistics, since these numbers are most readily available and can be easily translated into statistics for other devices.

Tape 2 consists of 8 files, as follows:

File No.	Approx. Size (2311 tracks)	Tape Records	DSNAME	Member Names	Contents
1	120	395	CMPLR3	AED1	AED-1 Compiler
2	67	179	AEDLIB3	*	Basic AED Text Library
3	16	38	FSPLIB3	*	"Fast Free" Text Library
4	64	214	AJR3	AJRV3	AEDJR System (complete)
5	56	152	AJRLIB3	*	AEDJR Text Library
6	72	246	MACRO	AEDMACRO	AED-1 Macro Pass
7	17	43	TRCLIB3	*	TRACE Debugging Text Library
8	142	348	VERSJV3	*	AED-1 Text Library

Each DSNAME given in the above table is preceded by the characters "USERFILE.M0846.0123.LOAD.AED." in order to comply with MIT's catalogue name conventions. 2311 disks are recorded at 3625 bytes of information per track, and 10 tracks per cylinder. All modules were recorded with file parameters BLKSIZE = 3072 and RECFM = U.

To use the tape, the user runs an IEHMOVE job to copy the desired tape files onto whatever direct access device he wishes except for the Data Cell (2321), whose blocking factor does not permit a BLKSIZE of 3072. To use the resulting Partitioned Data Sets, he uses the proper DSNAME and device in the DD card for his JOBLIB (to run the compiler, the Macro Preprocessor, or AEDJR -- tape files 1,4, and 6) or for his SYSLIB (to use one of the libraries).

In addition to the files contained on tape 2, certain other source libraries are required to run AED, and these are generated from tape 1. The specific source libraries required depend on the portion of AED that is to be used. For example, to use the AED-1 compiler, the assembler macro library from file 4 of tape 1 must be available to assemble the programs which are produced by the AED-1 compiler. File 6 (.INSERT files) will probably also be needed so that basic data structure definitions may be manipulated by standard symbolic referents in source programs.

3. Tape 3: CMS Modules and Libraries

Tape 3 contains the MODULE, TXTLIB, and MACLIB files of the CP/CMS version of AED. The tape is divided into four files:

File No.	Name1	Name2	CMS* Records	Tape Records	Contents
1	AEDV3	MODULE	8		AED-1 Compiler Passes
	AEDPH1	MODULE	133		
	AEDPH2	MODULE	79		
	AEDPH3	MODULE	117		
	AEDPH4	MODULE	29 (366)	866	
	AEDLIB3	MACLIB	44		Assembler Macro Library
	AEDLIB3	TXTLIB	102		Basic Run-Time Library
	AIOLIB3	TXTLIB	213		I/0 Run-Time Library
	TRCLIB3	TXTLIB	76		TRACE Debugging Library
	FSPLIB3	TXTLIB	74		Fast Free Storage Library
2	AJRV3	MODULE	165	733	AEDJR (complete version)
	AJRLIB3	TXTLIB	276		AEDJR Text Library
	AJR	TEXT	270		AEDJR Text Decks
3	AEDMACRO	MODULE	186	187	AED-1 Macro Pass
4	OSTOCMS	MODULE	3	102	Read OS tape into CMS Directory.
	OSTOCMS	TEXT	14		
	OSTOCMS	SYSIN	84		

Tape 3 is recorded in CMS TAPE DUMP format. Therefore, by attaching the tape and using the CMS TAPE commands (TAPE SKIP, TAPE LOAD, TAPE SLOAD, etc.) the files listed above may be read into the user's directory or into the installation's command directory. The user may thus select whatever portion of AED he may wish to use, and if file directory space is a problem, the needed files may be loaded from tape each time they are needed.

The four files fall into the following categories: The AED-1 Compiler and its run-time support libraries, the AEDJR System, the AED-1 Macro Preprocessor, and the OSTOCMS utility. Use of the first three is described in the related user documentation.

The OSTOCMS utility program (file four) permits the OS Source Tape contents to be read selectively into the user's file directory. OSTOCMS is designed to read from OS labeled or unlabeled tape in blocked or unblocked format, with a maximum blocking factor of 14,400 bytes-per-record. The tape is rewound when the OSTOCMS command is given, and

* Numbers are approximate. Actual files may be slightly larger.

the program begins by reading the tape label and printing it on the console, or by printing "unlabeled tape". The program is designed to create a single file for each command issued, and to read the entire amount specified, including all of the ADD cards in the file created.

The OSTOCMS command format is as follows:

```
OSTOCMS name1 name 2  $  (options)
```

All arguments are optional, and if the command OSTOCMS with no argument is given, the entire file is read into the user's directory. Name1 and name2 are the two names of the file to be created. The default values for name1 and name2 are OS and TAPE, respectively, so that if no file name is given, a file named OS TAPE is created, and if one name is given, a file named "name1 TAPE" is created. The (options) arguments following the "$" may be given in any order, and are as follows:

```
SKPn
TAPn
FROM = prog.name
TO =   prog.name
```

SKPn tells OSTOCMS to skip n files before reading the tape. TAPn tells OSTOCMS which tape to read from, where TAP2 is the default case, and the tape unit numbers are those assigned at the local CMS installation. FROM= prog.name determine where to start reading within the tape file, where prog.name is the name appearing on the IEBUPDAT ADD card. If no FROM = option is specified, OSTOCMS begins at the beginning of the file. Similarly, TO= prog.name tells OSTOCMS where to stop reading, and if the TO= option is omitted, reading continues to the end-of-file. OSTOCMS includes the TO= program in the CMS file created so that, to read a single program from a file, the same prog.name is used for the FROM= and TO= arguments.

4. Tape 4: CMS Text Files

Tape 4 contains all of the Text (object) programs for the AED System, and is written in the TAPE DUMP format of CMS. The tape contains the following 7 files:

File No.	Name1	Name2	Tape Records	Contents
1	AJR	TEXT	277	AEDJR System
2	LIB1	TEXT	369	AED Library, Part 1
	LIB2	TEXT		AED Library, Part 2
	LIB3O3	TEXT		AED Library, OS Dependent
	LIB3CMS	TEXT		AED Library, CMS Dependent
3	AEDMACRO	TEXT	313	Macro Pass, System Independent Part
	MACROOS	TEXT		Macro Pass, OS Dependent Part
	MACROCMS	TEXT		Macro Pass, CMS Dependent Part
4	P1CMS	TEXT	628	AED-1 Compiler, Pass 1, CMS Dependent
	P1OS	TEXT		AED-1 Compiler, Pass 1, OS Dependent
	P123	TEXT		AED-1 Compiler, Progs. Common to Passes 1,2,3
	P12	TEXT		AED-1 Compiler, Progs. Common to Passes 1,2
	P1	TEXT		AED-1 Compiler, Pass 1
	P2	TEXT		AED-1 Compiler, Pass 2
	P3A	TEXT		AED-1 Compiler, Pass 3, Part A
	P3B	TEXT		AED-1 Compiler, Pass 3, Part B
	P4	TEXT		AED-1 Compiler, Pass 4, Alarm Reporting
5	FSPLIB3	TEXT	75	"Fast Free"Package
6	TRACE	TEXT	77	TRACE Debugging Package
7	SHOIT	TEXT	66	SHOWIT AEDJR Example

Tape 4 contains all text files for the AED System, and is designed for CP/CMS only. The text libraries contained on Tape 2 are sufficient for compiler and AEDJR link-editing when creating new version of these modules, so no additional OS text tape corresponding to Tape 4 is necessary.

C. VERSION 3 BOOTSTRAPS TO OTHER COMPUTERS

Concurrent with the bootstrap of AED from the MIT 7094 time sharing system to the IBM 360, work was in progress by other cooperating groups to effect bootstraps to two other computers; the Univac 1108 and the GE 645. Neither of these efforts, which are reported below, was a primary responsibility of the MIT project, although as much assistance was given as possible. A third effort to bootstrap AED to the Digital Equipment Corporation PDP-10, was conducted entirely outside the MIT project by Codon Computer Utilities.

1. Univac 1108

Subsequent to the participation of Mr. Robert Coe in the AED Cooperative Program in 1965/66, the United Aircraft Corporation (UAC), Hartford, Connecticut, took the lead in attempting to bootstrap AED to the Univac 1108 computer. This resulted in several 1108 versions, culminating in late 1968 with limited distribution by UAC of a Version 2 AED/1108 which ran under a modified EXEC2 operating system.

In June, 1968, the Univac Corporation established a direct interest in 1108/AED, with a joint MIT/Univac goal of creating a new 1108 bootstrap completely compatible with the Version 3 AED/360. Univac assigned Mr. Richard Gluckstern to the AED Cooperative Program, and by the end of his one-year term in June, 1969, which coincided with the termination of AED Project activity, Mr. Gluckstern had become well acquainted with the bootstrap process, and had made substantial progress. Also, the needed files and programs were made operational on the IBM 360, so that the 1108 work was no longer dependent on access to the MIT 7094 and could be done at Univac.

In February, 1970, Univac announced the availability of a new Version 2 AED which operates under the standard EXEC2 operating system (instead of the modified EXEC2 required by the UAC versions). Work is continuing at Univac on a Version 3 AED/1108 that will operate under the EXEC8 operating system.

2. GE-645

Starting in the fall of 1968, an effort was started to bootstrap AED to the Project MAC GE-645 computer under the Multics time-sharing system. Since the Computer-Aided Design Project did not have the resources (time and people) to carry out this work by itself in addition to its 360 bootstrap work for its Air Force sponsor, a team of people from Project MAC and other MIT projects was assembled to do the bootstrap, with the assistance of key project personnel. Work proceeded rapidly, and by early 1969, a half-bootstrap to the 645 was working on the 7094. This produces a 645 assembly code, which can then be assembled and run on the 645.

By June, 1969, work was about 60-percent complete on a first-phase full-bootstrap of Version 3 AED that would run entirely on the GE 645, but would not take advantage of the Multics paging mechanisms (a later second-phase effort was planned to produce a paged version). These full-bootstraps to the GE 645 were not completed, however, because of the time demands on the AED project staff members to complete AED/360, their subsequent termination from MIT and a realignment of Multics priorities. It is hoped that these bootstraps will again be taken up when circumstances permit.

D. AED DOCUMENTATION

1. Documentation Problems

A valid criticism voiced by many "outside" users of field-trial versions of AED systems over the past few years is that the documentation provided was not well organized, difficult to work with, and in some areas incomplete. Because of the complexity of high-level computer languages and systems, the preparation of the various levels of tutorial material and user's manuals needed for both beginning and experienced programmers is a major task, even after the language and system are stable and well defined. Where the language and system are under a state of continual change, as they are during a fast-paced development program such as AED, extensive formal documentation of interim phases is next to impossible, both because much of it would rapidly become obsolete (perhaps before even being completed), and because the personnel qualified to prepare the documentation are the same ones needed for the next stage of development. Of course, the interim documentation issue would not arise if the system were kept "in house" until development were completed. However, a most valuable aspect of a development program is the feedback obtained from outside users of interim versions. Thus a paradox arises.

Early in the AED development program, the decision was made to obtain as much feedback as possible by distribution of interim systems and subsystems to interested outside groups, and we believe this decision was correct. Every effort was made to provide accurate technical descriptions, but for the reasons stated previously, these usually took the form of rather specific memoranda, each describing a particular new feature or system change. Thus the total system documentation, known as the "AED User's Kit", took the form of a loose-leaf collection of memoranda and "system flashes" that eventually totaled some 1000 pages. This contained all the necessary information, but in an unorganized manner which made it difficult document to work with, particularly for those without day-to-day personal contact with the AED development group. The perserverence shown by the many who did learn to use AED primarily from this documentation spoke well of the interest in AED, but it was clear that the chaotic state of the documentation was a deterrent to wider use of AED. Thus a major documentation effort was mounted in the fall of 1968, concurrent with the preparations for the Version 3 release.

2. The Final Documentation Effort

Recognizing that professional writing help was needed to complete the task of documenting AED, a subcontract was let by the MIT Computer-Aided Design Project to Cambridge Computer Associates, Inc., of Cambridge, Massachusetts. The unstinting efforts of CCA personnel, working closely with C.G. Feldmann, have transformed the "AED User's Kit" into a two-part AED-0 Programmer's Guide (Report ESL-R-406), the table of contents of which is shown in Table II. The Guide is designed as a user reference manual and as such, each chapter discusses one aspect of the AED language or subroutine library in complete detail. Part 1 is a complete description of the AED-0 language proper--including descriptions of several subroutine packages (e.g., optional procedure call arguments) which extend the features of the language beyond the syntactic forms derived from Algol-60 syntax. Part 2 presents further subroutine packages useful as building blocks for general software system contruction. Because the report is a reference document on the use of AED, the above arrangement of material is not ideally suited to learning the AED-0 language, thus a tutorial document was also needed.

Working closely with D. T. Ross, CCA personnel also transformed the edited transcripts of the first 12 lectures of his special MIT graduate course "Software Engineering", given in the 1968 Spring Term, into a companion report, Introduction to Software Engineering with the AED-0 Language (ESL-R-405), the table of contents of which is shown in Table III. This report takes a tutorial approach to the description of

Table II

Contents of Report ESL - R - 405, " Introduction to Software Engineering with the AED-0 Language"

Table II (cont'd)

Table II (cont'd)

Table II (cont'd)

Table II (cont'd)

Table II (cont'd)

Table III

Contents of Report ESL-R-406, "AED-O Programmer's Guide"

CONTENTS - PART I

Table III (cont'd)

Table III (cont'd)

Table III (cont'd)

Table III (cont'd)

Table III (cont'd)

most of the basic features of AED programming, and is recommended as a starting point. Once the reader has basic familiarity with AED methods, the more advanced topics treated only in this report can be easily learned. Simpler aspects of each topic are treated first, so the reader may use basic features without knowledge of full details.

Taken together, these two reports total over 550 pages, and constitute the revised, greatly improved documentation for the Public AED Release. All of the writing had been completed in draft form by the time of the departure of the AED project staff from M.I.T. (following the Version 3 release in July, 1969) and about two-thirds of the material had been edited and final-typed on galleys, using IBM Magnetic Tape Selective Typewriter and Composer (MT/ST and MT/SC) equipment. However, much work remained in editorial review and rewrite of remaining material by CCA, proofreading of all material by both CCA and the authors, galley pasteup and correction, preparation of indices, final typing of one-third of the material, completion and checking of the large number of drawings, and the printing process itself. Report ESL-R-405 was distributed on December 23, 1970, and Report ESL-R-406 (AED-0 Programmer's Guide) on March 6, 1970.

Other documents relating to the application of AED were also issued during the reporting period. These will be discussed in subsequent chapters in connection with the descriptions of this work.

CHAPTER IV

APPLICATION, LANGUAGE, AND COMPUTATION STUDIES

In addition to the main effort of the project in preparing for and completing the AED/360 compiler bootstrap, other parallel AED efforts were under way, particularly during the first of the two contract periods covered by this report (1 May 1967 to 31 November 1968, under Contract F33615-67-C-1530).

The AED compiler relies heavily upon the other AED tools — the library of subroutine packages, and the AEDJR system for language definition (used for the AED-0 language definition). During the reporting period, all of the subroutine packages and the AEDJR system were compiled and checked out on the new compiler versions, and additions and corrections were made to enhance their capabilities, as described in Sections A and B. The final versions of these are included in the Version 3 AED/360 release (see Chapter III).

Other efforts included the SHOWIT System (Section C), which is a practical example of the application of the AED approach, and which is included in the Version 3 release; the Syntax Definition Facility (Section D); the Equilibrium Problem Solver (Section E); and a study of graph models for parallel computation (Section F). Each of these latter studies is covered in a separate published report, so they will only be summarized here.

Another AED application study (CADET) was only partially completed, and had to be set aside at the beginning of the second contract period (1 December 1968) because it was outside the agreed-upon tasks. Since this work was not separately reported, it is covered in some detail in Chapter V.

A. AEDJR REVISIONS

In addition to compiling the 84 programs in the AEDJR system for the 360 through the new compiler versions and checking out the result, the entire AEDJR system was reworked for machine independence. While in the process of reworking the AEDJR source programs, some additions were also made. Two new AEDJR commands, OPNFIL and CLSFIL, were added to allow the user to write lengthy console output onto a remote file rather than the slower, on-line console. Also, the program which sets up the AEDJR "LIKE string" data structure was changed to place the elements in the string in the order given by the user in the AEDJR source program rather than in the ascending-core-location order used previously. This permits the user to organize the LIKE statements in the order that matches are most likely to occur, thus increasing processing time efficiency during a run.

A parallel effort was also started to activate the AEDJR MARK command on the 360 and 1108, so that processed AEDJR vocabularies may be punched out as a deck of macro calls and then assembled into binary form for production runs of debugged, completed vocabularies. Up to the end of the reporting period such vocabularies (e.g., the AED-0 vocabulary) had been processed via MARK on the 7094, and the resulting macro decks then assembled on the 360 or 1108. A certain amount of reworking of the MARK programs was necessary to make them machine independent, and this process was started during November. While these revisions were being done, additional flexibility was added to allow user specification of entry points in the MARK deck, so that the AEDJR vocabulary table may be referenced at specific points by user programs.

Work also continued on the AEDJR user's document, which was elaborated by examples.

B. SUBROUTINE PACKAGE EFFORTS

In addition to the considerable efforts involved in correcting bugs, updating the subroutines into the libraries, and compiling and re-compiling the several hundred subroutines in the AED library, many new subroutines and reworked versions of old subroutines were written and checked out in a continuous effort to improve the AED software system tools. The remainder of this section briefly describes some of these efforts.

The free Storage Package for dynamic storage allocation during execution is one of the most critical of the subroutine packages. It underwent a new analysis and re-writing during the reporting period.

Many revisions were made in the Input-Output Buffer Control Package (IOBCP) which is a vital link between AED and its machine environment. Heavy usage on several different computers continually suggested changes, so that by the end of the reporting period, a vastly better package was available.

The Delayed Merge Package (DMERGE) accepts data in an input order and outputs the same data in a different, output order under flexible user control. The package will be used in the next analysis of the Compiler's Second Pass (AED-2 final compilation phase), and may be used by anyone having a similar data-shuffling problem. During the reporting period, DMERGE was converted and checked out on the 360. Several machine-independent areas were reworked during the conversion, and the revised DMERGE source files are system and machine independent. Since DMERGE makes heavy use of IOBCP (Input-Output Buffer Control Package), it afforded a real test of IOBCP, and suggested some minor changes.

A SNAP package for OS/360 debugging was written, and debugging was completed. The package permits selective run-time dumps.

A ".B." package to handle binary strings in a way similar to character strings was written and debugged. A memorandum was written.

A simplified CPRINT package was written to print .C. strings without using the full ASEMBL package.

A 360 version of the Alarm Package was compiled and debugged. The new version further breaks down the checking and reporting phases for greater flexibility.

C. THE SHOWIT SYSTEM: AN EXAMPLE OF THE USE OF THE AED APPROACH

Since 1959 the M.I.T. Computer-Aided Design Project has been engaged in a program to establish the techniques of implementing man-machine systems for computer-aided design. From the beginning it was recognized that "design" is but a special term for some ill-defined type of problem-solving; no distinctive features are reflected in a system for design versus a system for general problem-solving. However, the field of man-machine problem-solving is much too broad to permit a single system to be used for all applications. Many systems are needed, each of which must

1. use the specialized jargon of its particular field of application,

2. require little or no knowledge of computer programming to be used effectively,

3. be evolutionary to adapt to the changing needs of its users, and

4. be created and maintained by the users themselves or by skilled local staff who are in intimate contact with the users.

For these reasons the Project's efforts have not been directed toward a single computer-aided design system, but rather toward a system for making systems. Actually, what has evolved is a "system of systems for making systems", with an orderly method for applying it. This collection of concepts and working tools we refer to as the (AED) Automated Engineering Design approach.

The SHOWIT System, which originated as an unscheduled demonstration of AED cpapbilities at the Second AED Technical Meeting in January, 1967, is a tutorial example of the application of the AED approach, with particular emphasis on the facilities of the AEDJR parsing processor for implementing new languages. The SHOWIT language is a subset of the Iverson language, chosen at the time of the January, 1967 AED Meeting.

By using AEDJR as the framework for the new system, the programmer can implement quickly all the procedures and grammar rules. In particular, the initial plateau of SHOWIT acts as a subsystem of AEDJR. By means of three successive commands to AEDJR, the following events can be caused:

1. The grammar rules for the entire SHOWIT language are read in and made active.

2. A special set-up procedure, written by the programmer, is invoked.

3. Any specific program, written in the SHOWIT language, is executed.

The lexical processor for the SHOWIT System is a specially constructed RWORD machine.* Its item-building rules conform to the designer's specifications for the SHOWIT language. In particular, the lexical phase identifies and discards comments written by a user in his input message. The RWORD machine is invoked — that is, a request is made for a new item to be extracted from the input — by a special procedure written by the programmer to replace the standard one provided in AEDJR. The programmer's procedure is itself called by the First-Pass Algorithm each time a new input item is needed to continue the parse. A second RWORD machine is used in the SHOWIT System to read in data values typed on-line by a user. This machine accepts numeric items written in integer, decimal, or E-type format.

The SHOWIT System has been documented in Technical Memorandum ESL-TM-394, by J. R. Ross and D. T. Ross, June, 1969, and is included in the AED/360 system release.

D. SYNTAX DEFINITION FACILITY (SDF)

One of the central topics in the computer science field since the introduction of the Algol report in 1960 has been what can be called the language definition problem. The problem is, simply, to find a "good" method of defining or describing a computer programming language and of producing a processor for it. Of course, there are probably as many meanings for "good" as there are uses or users of computer languages. One use of a language definition is for people to read to learn the language. In this case "good" will probably mean that the definition is easily readable and understandable. Another similar use would be as a reference for a person who knows the language. Here "good" would mean not only readable but also concise and complete. A third important use of a language definition is as input to a computer program to produce a compiler for the language. For this use "good" might

* A description of RWORD is given in Appendix B, which is a reproduction of a technical paper in the Communications of the ACM, Vol. 11, No. 12, December, 1968.

mean precise. Also, since the designer of the language must write the definition, "good" to him implies "easy to write".

It is possible to produce several definitions of the same language — one to serve each purpose — but, this has several disadvantages. First, to produce several different definitions can be expensive, since it involves duplication of effort. A second and more serious problem is that the definitions may not be precisely equivalent. For example, the old method of compiler construction starts from a concise, "for-people" description of the language. The compiler is then hand-coded from the original description. The chances that the hand-coded compiler implements exactly the full intent of the original description are very small.

The various requirements of a language definition system — readability, understandability, conciseness, completeness, precision — are often conflicting. But, as mentioned above, it is undesirable to have several definitions of the same language, so it is necessary to find a good compromise. The problem is essentially to develop a practical system in which a programming language can be described in a precise yet readable form which can be processed automatically to produce a compiler for the language. To be practical, the system must be convenient and economical to use. It must handle the types of constructions usually found in programming languages, and it must produce an efficient processor for the language.

The language definition problem was studied in a thesis research by R. S. Eanes, resulting in a system called the Syntax Definition Facility (SDF). This is an interactive system which allows the designer of a computer programming language to define the syntax of his language in a relatively simple natural meta-language. From this syntax definition a set of tables for driving a general parsing algorithm are produced. If the system detects possible ambiguities or inconsistencies in the definition supplied by the language designer, it will report them and try to indicate the source of the problem. The system includes test and debugging facilities to aid the language designer.

The SDF meta-language allows the language designer to specify the syntax of his language by writing a series of <u>sample statements</u> which are marked to indicate how they should be parsed. Each sample statement specifies the "kind of value" or semantic type of a construction in the language. By the underlying principle of phase substitution, which allows any construction to be substituted for any other construction of the same semantic type, the small number of sample statements induces a complete language definition allowing statements of arbitrary size and complexity. A syntax definition in the SDF meta-language is somewhat similar to a Backus-Naur Form or context-free grammar definition, but it is more readable and easier to produce. The algorithm used by the system to produce a parser from the meta-language description is a synthesis of the precedence techniques and the AED Language definition systems.

The thesis has been issued as Report ESL-R-397, "An Interactive Syntax Definition Facility," R. S. Eanes, September, 1969.

E. EQUILIBRIUM PROBLEM SOLVER (EPS)

"Equilibrium Problem Solver" is an expression suggested by the fact that a mathematical physics, boundary value problems generally arise when continuum models are employed in the analysis of physical systems at equilibrium, such as temperature distributions in structures, air flow through orifices, etc. The EPS system is a computer program with an input language especially tailored to these types of applications. Specifically, when the development of EPS first began in the fall of 1964, it was intended that EPS solve two-dimensional boundary-value problems for linear elliptic systems of second-order partial differential equations, and it was hoped that users would be able to obtain solutions to particular problems by supplying only essential information describing governing equations, boundary geometry, and boundary conditions.

In a sense, EPS has both overshot and fallen short of its original goals. It has overshot these goals insofar as it is able today to treat a class of problems which is much broader than, although not so rigorously definable as, the class for which it was initially intended. On the other hand, EPS is not the "fully automatic, numerical assistant" once envisioned. With the benefit of hindsight, it is possible to say that both of these results were inevitable. The class of problems which EPS was meant to solve is itself broad enough to indicate the use of numerical methods which, for maximum flexibility, are best implemented in a semiautomatic form. Thus, in particular, it is necessary in EPS for the user to define a discrete model for this problem, in the form of a finite-difference lattice, in addition to the various parameters which are mathematically essential. At the same time, however, the many variables left open by the class of problems for which EPS was intended necessitated certain general-purpose, algebraic capabilities which make possible many important extensions. For example, it is possible to have EPS compute revised problem or lattice parameters using results from previous solutions. Consequently, iterative procedures can be effected which permit the treatment of boundary-value problems with nonlinearities, free surfaces, or undefined parameters. Further, the general-purpose feature of the system can be exploited in an independent manner, and enable the user to specify and have executed algorithms which have nothing to do with the solution of boundary-value problems

When EPS was started in 1964, AED was not sufficiently developed to make it attractive for use as the programming language for what was thought to be a short-term thesis research program. Thus the first version of EPS, completed in 1966, was not programmed in AED. During the reporting period, EPS was completely reprogrammed in AED for the 7094 time-shared computer at M.I.T., and now exists in two forms — a typewriter version, and a version which uses graphical input-output on the ESL Display Console. A movie demonstrating EPS was completed (except for a sound track), and a user's manual was completed and published as a joint report with Project MAC, which provided a considerable share of the support for the EPS work. The user's guide, which will be an appendix of a forthcoming doctoral thesis is:

> Report MAC-TR-62/ESL-R-395, "EPS: An Interactive System for solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation"
>
> C. C. Tillman, Jr. —— June, 1969

F. GRAPH MODEL FOR PARALLEL COMPUTATIONS

Present computers operate on a serial basis; that is, parts of a problem are worked on one at a time in a predetermined order by a single arithmetic unit. Since further major advances in circuit speed seem unlikely, there is great interest in computer configurations which permit parallel processing; i.e., simultaneous processing of different parts of a problem by multiple arithmetic units. One of the major difficulties in such processors, however, is that of structuring a computational process so that interdependencies between various computational steps (such as the requirement that the output of one step be the input of another) can be properly handled.

This problem was studied in a doctoral thesis research by J. E. Rodriguez, completed in September, 1967. The thesis has been published as a joint ESL/Project MAC report, "A Graph Model for Parallel Computation," Report ESL-R-398/MAC-TR-64, J. E. Rodriguez, September, 1969. The results of this study are best summarized by the abstract, reproduced below:

> "This thesis presents a computational model called <u>program graphs</u> which make possible a precise description of parallel computations of arbitrary complexity on nonstructured data. In the model, the computation steps are represented by the <u>nodes</u> of a <u>directed graph</u> whose <u>links</u> represent the elements of storage and transmission of data and/or control information. The activation of the computation represented by a node depends only on the control information residing in each of the links incident into and out of the node. At any given time any number of nodes may be active, and there are no assumptions in the model regarding either the length of time required to perform the computation represented by a node or the length of time required to transmit data or control information from one node to another. Data dependent decisions are incorporated in the model in a novel way which makes a sharp distinction between the local sequencing requirements arising from the data dependency of the computation steps and the global sequencing requirements determined by the logical structure of the algorithm.
>
> The concept of the <u>state</u> of a program graph is introduced and it is proved that every program graph represents a <u>deterministic</u> computation, i.e., that the final state of each computation started from the same

initial state is unique. Computations which do not terminate properly are defined in terms of the concept of hang-up state. Methods of analysis are developed and necessary and sufficient conditions for the absence of hang-up states are obtained. These conditions are interpreted in terms of the structure of the graph and the manner in which the decision elements are imbedded in that structure. Finally, an equivalence problem for program graphs is formulated and a solution to this problem is presented."

CHAPTER V

DATA MODELING (CADET)

A. INTRODUCTION

The primary goal of the Computer-Aided Design Experimental Translator (CADET) portion of the computer-aided design effort was to achieve an over-all systematic model of the man-machine problem-solving process.* In working toward this goal, many of the packages developed in other parts of the Project (e.g., AEDJR, Free Storage, String Package, BCORE) were utilized, as well as a number of new techniques and packages developed specifically for this purpose. This chapter describes the progress that was made in developing a new technique in data modeling for the CADET programs.

One of the important problem areas in computer-aided design is that of making a model to represent the object or system being designed. By model we mean a numerically encoded symbolic representation stored in the computer memory. The model must contain all pertinent data as well as all the useful relationships between various data. "Good" mechanizations for this information will allow for efficient processing of the model (growth, modification, analysis) and will not be wasteful of this computer storage by making unnecessary duplication of identical or similar elements of the model. Plex theory** provides a general approach to modeling and gives much of the strategy of data modeling. For example, plex theory shows how to set up a semantic package to perform the primary operations for growth and analysis of the data model, and also how to design a search function called a "mouse" which locates the data

---

* Investigations in Computer-Aided Design for Numerically Controlled Production, Interim Report IR 8-236-VI, 1 June - 30 November 1966, p. 29.

** Investigations in Computer-Aided Design for Numerically Controlled Production, Final Report for the period 1 December 1959 - 3 May 1967, Report AFML-TR-68-206 (M.I.T. Report ESL-FR-351), Chapter III.

stored in the model, as required input for the primary operations. However, there is no "cookbook" for resolving the tactical question of how to mechanize the data model, the semantic package, and the various mouse functions. One useful set of tools for the system programmer is the facilities of the Generalized String Package, but although the basic techniques of the String Package are appropriate, there are insufficient side constraints in their use to be of much tactical assistance. The following sections describe efforts to encompass more of the total modeling problem in a single cohesive scheme, to give a still higher level of approach than the Generalized String Package provides.

B. DESCRIPTION OF POLYFACE

To study the tactical problems of modeling, it was necessary to have an application area which may be studied in detail. The area chosen was "Polyface", which was first presented to the attendees of the Second AED Technical Meeting in January, 1967 as an example of the utilization of the String Package for semantic package design. Polyface is concerned with modeling "objects" composed of polygonal faces joined along their edges. Thus Polyface uses only very simple geometry, and the modeling requirements are almost purely structural. We expected that, because of this structural purity, most of the techniques derived for the polyface example would carry over to more general modeling contexts. Eventually, of course, we intended to apply our techniques to some "real life" design applications.

We chose to model objects constructed from planar faces whose boundary edges are straight lines. For the remainder of this discussion, the word object always refers to such a construct. The set of programs written to generate and modify objects composed of these polygonal surface elements is called Polyface, or the "Polyface Package." Four useful structure-changing operations are shown in Fig. 12, and they have been programmed along with their four inverse operations.

The objects formed by these operations are modeled using rings of pointers to associate together the various face, edge, and point beads which make up the entire object. An example is diagrammed

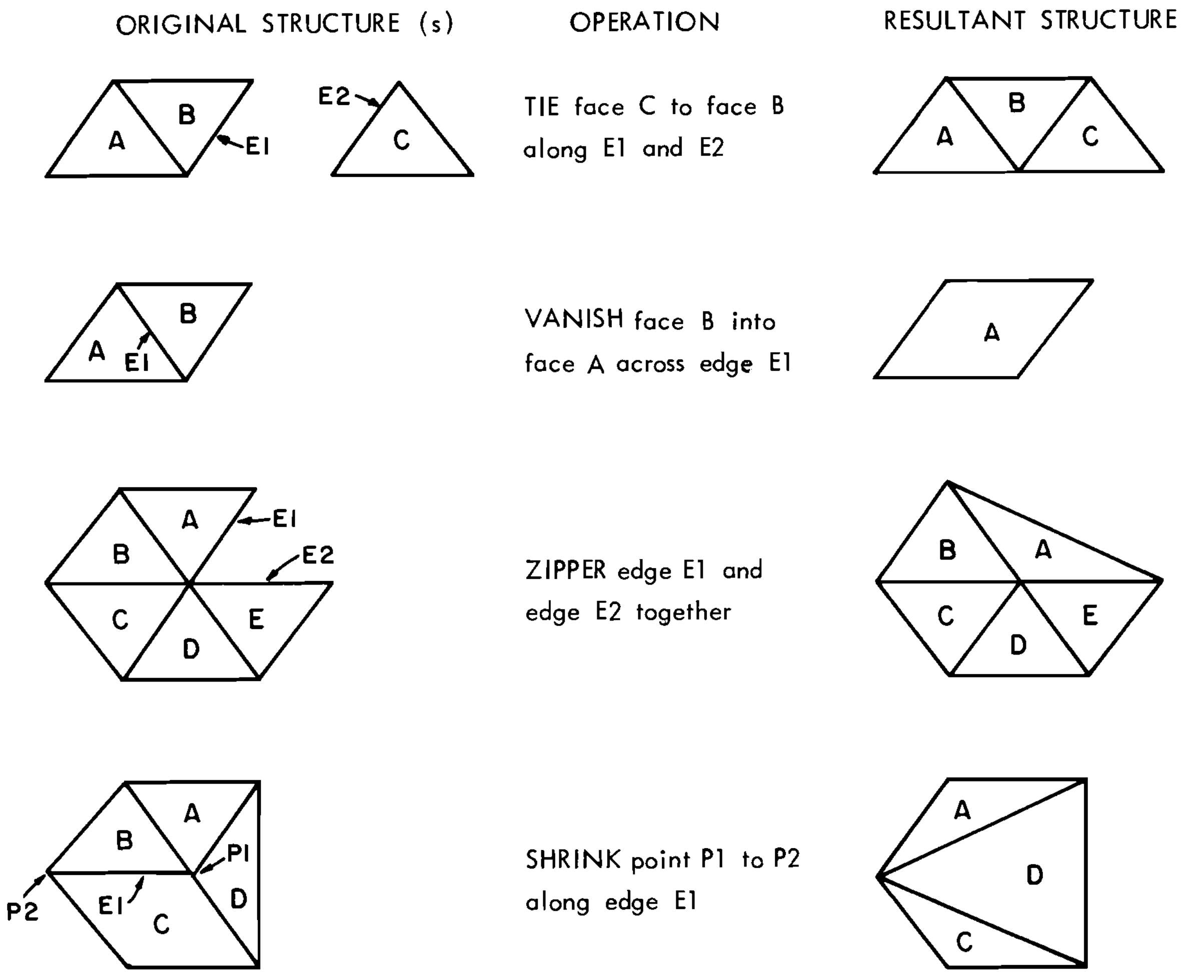
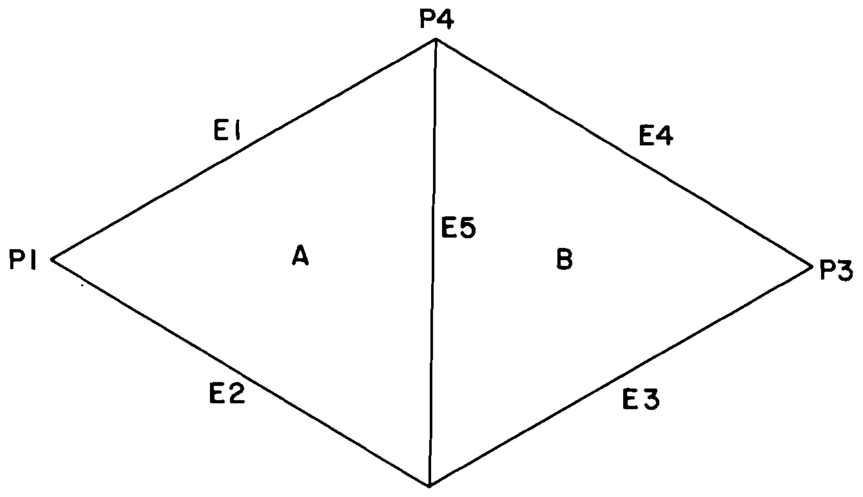

ORIGINAL STRUCTURE (s)
OPERATION
RESULTANT STRUCTURE
A
B
EI
E2
C
TIE face C to face B along E1 and E2
A
B
C
B
A
EI
VANISH face B into face A across edge E1
A
B
A
C
D
E
EI
E2
ZIPPER edge E1 and edge E2 together
B
A
C
D
E
A
B
PI
P2
EI
C
D
SHRINK point P1 to P2 along edge E1
A
D
C

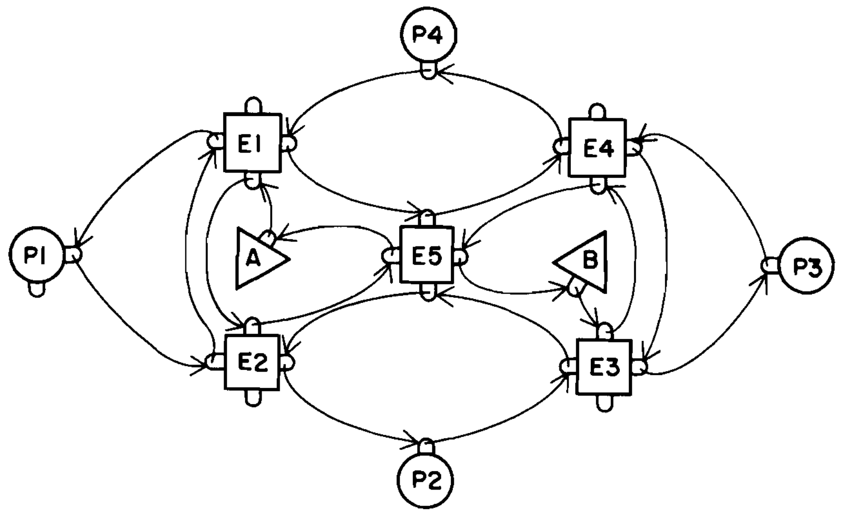
Fig. 12 Polyface Operations

in Fig. 13 . Notice that there are rings which associate faces with their boundary edges, and also rings which associate points with the edges they bound. Objects are modeled as a bead with a name component, entered in a symbol table. Each object bead has a pointer to one of the face, edge, or point beads in the topological structure. This then serves as the entry position for a mouse function which can follow the ring pointers in some sequence while it calls an action function. We have studied the following types of mouse functions:

1) Spiral Mouse - follows face-edge rings beginning at a given entry position. In running a large structure, the sequence of faces lie on ever expanding curves about the entry position, hence the name "spiral",



(a) Two-faced object with named elements



(b) Ring model of object (a)

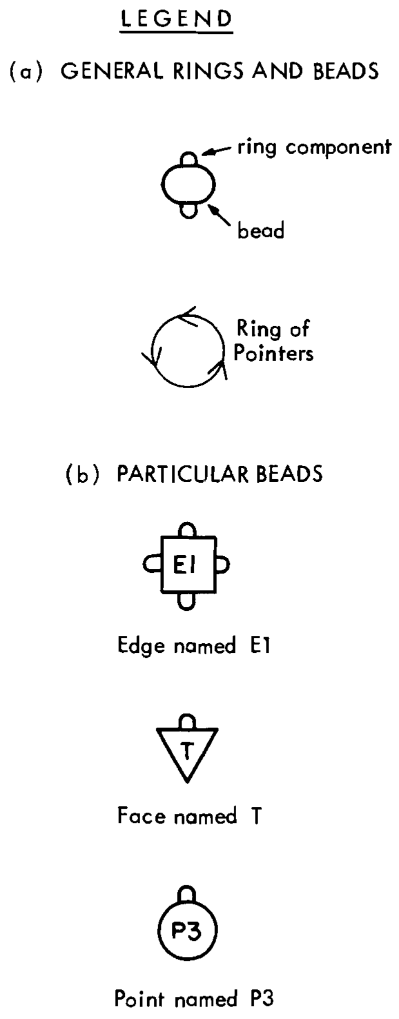


Fig. 13 A Polyface Model (Topological)

2) SCAN.RING - starts at any bead on a given ring and goes around the ring once,

3) Boundary Mouse - goes from the entry position to the boundary of the object — then makes a circuit of the boundary edges and points,

4) Steerable Mouse - the direction of its next step is controlled by the function it calls.

In the work which was completed, only mouse functions 1) and 2) were used for plotting a picture representing the object, and for performing the eight polyface operations (i.e., TIE---) respectively.

1. The Initial Polyface Program

In the original Polyface program the method used to carry out the various operations consisted of three phases:

1) Temporarily attach new beads specifying the changes to be made to the topological structure of the object(s) being changed.

2) Send a mouse around each of the above objects, carrying a copy function which makes an entire new copy, incorporating all the changes so indicated.

3) Destroy the original objects if desired.

This technique, although exemplifying a straightforward application of Plex Theory, has two disadvantages when considered as a potential data modeling tactic. The first disadvantage is that much computer time must be used in making duplicate copies of large structures, where only a few localized changes must be made to carry out the Polyface operation. The second disadvantage is that either the previous structures are lost in step 3) or large amounts of storage are required to contain all intermediate structures during the design process. The seriousness of this disadvantage is seen when one realizes that the designer would like the security of being able to "back up" to a previous state if his latest trial is not successful. He would also like the flexibility of being able to try several designs and then choose the best. A technique for eliminating these disadvantages is described next.

C. COMMON SUB-EXPRESSIONING

1. Call Mechanisms

The most widely used technique permitting compact storage of redundant information is based upon the subroutine definition and call mechanism. This technique may take many forms, but always includes the concept of supplying a single master definition incorporating selected changeable parts indicated by dummy arguments. Specific instances of this generic definition are then obtained, with desired modifications, by means of a call with specific argument values. The body of the subroutine definition is not actually copied (as in the formal definition of an ALGOL 60 procedure) but rather is directly referenced as in most mechanizations of a subroutine call (e.g., FORTRAN). The concept is equally applicable to a call on a structural body of data serving as a computational model. Thus a single generic model of an object can yield any number of instances by calling the data body a sufficient number of times.

Although quite effective, especially since the specific implementation of the call mechanism may take many forms, this concept suffers from the defect that the changeable parts (arguments) must be known beforehand and are rigidly fixed at the time of definition. A much closer match to the needs of a creative user of a system would result if one were able to say (regarding either programs or objects) that a new object shall be just like the object or objects composing it except for a number of arbitrary variations not previously specified. In this way all common portions of the data structures modeling the objects would be shared, and the user has complete freedom to alter any feature on the spot.

Such a sharing scheme has been devised for Polyface, yielding a model from which any of the desired variants can be extracted by a mouse function. By a sequence of operations, we form one all-inclusive data structure which models simultaneously all objects created by those operations. To select a specific object, (i.e., make a call on that object), we tell a mouse we want object ϕ. The mouse then runs through the structure and reports back all of the beads that are consistent with the ϕ assumption. In effect, the call mechanism occurs not in one place, but is spread out into the structure in the form of "variation beads" attached to each piece of data or structure in which the object ϕ differs from the master objects in terms of which it is defined. New objects are given new names, called object codes which are combined with previous object codes to form variation codes. Variation codes identify the variation beads which were created to form the new object.

In order to process this data structure, a mouse carrying an action function is given the object name, say ϕ, and an appropriate starting position. The mouse tests each new bead it comes to to find if a ϕ variation exists. If so, it is used. If not, the master (i.e., no variation) is used. As newer objects are created, which may involve variations of previous variations, a stack of object codes showing the nesting of sub-objects within a given object is generated. An example is the "trestle" shown in Fig. 14. The master triangle T was TIE'd to a call on itself, T_c, forming object $\phi 1$. A call on $\phi 1$, called $\phi 1_c$ was then TIE'd to $\phi 1$ itself, forming object $\phi 2$. Variations of particular master triangle beads are labeled $\phi 2$-$\phi 1_c$-T_c, $\phi 2$-$\phi 1$-T_c, etc. depending upon the sequence of construction operations

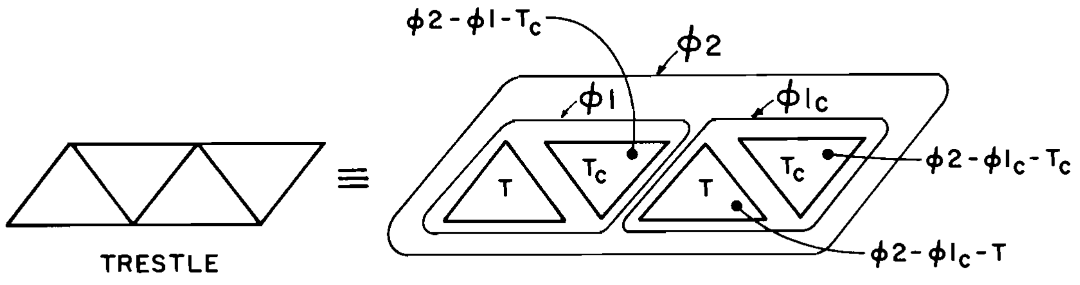


Fig. 14 Nested Polyface Structure Showing the Construction Sequence

which created the trestle. Each feature of the complete trestle has an appropriate variation code and thus may be extracted by the mouse.

2. Mouse Algorithm

Variation codes provide a unique static naming scheme for both master and variation beads in the data structure. As a mouse follows various pointers through the data structure, it crosses and recrosses the virtual boundaries of the nested structure indicated in Fig. 14. The general position of the mouse within these boundaries is given by a collection of code characters which show the nesting of sub-objects. Such a collection of sub-object codes is generated by the mouse as it moves about the object, and is called the context code of the mouse.

A context code includes a complete description of the mouse's position within the particular object ϕ through which the mouse is running. The mouse uses its context code at each step to locate the "best" variation bead. The best variation may have a variation code the same size[*] as the context code, indicating that it was created when ϕ was constructed. If no matching variation code of the same size is found, then the best variation bead is the longest one which is shorter than the context code and matches the largest number of the low-order characters of the context code. Such a bead was created for a sub-object of ϕ at an earlier time, and its code was not changed by later operations which led to the creation of ϕ. Variation beads whose variation codes are longer than the context code belong to objects of which ϕ is a sub-portion. Thus they are ignored by the mouse.

3. Better Encodings

As the previous sections indicated, both variation codes and context codes are collections of sub-object codes. This "stack" form of identifying the variation beads would require considerable storage at one sub-object code per memory word. Packing several codes per word would reduce the storage, but would also require the mouse to use much processing time to unpack the codes as it searches for the best match. Even with one code per word, the mouse would perform a separate comparison of each code in the bead with each code of the context code stack. Therefore some sophistication in the mechanization of codes is appropriate. Indeed, the only restrictions on the coding scheme which constrain the mechanization are: 1) the variation codes must be unique, i.e., a given bead must not have more than one variation with a particular variation code, for otherwise the mouse cannot determine which variation to use, 2) the variation code of a bead must "include" the variation code of its parent. This is necessary to show the genealogy or nesting of sub-objects upon which the concept of "best" depends. (It also provides an efficient tree searching algorithm.)

We have devised several ways of using a single binary string to encode stacks of "object names." The use of a single binary string reduces the storage space used and also permits several "object names" to be compared simultaneously, using full-word binary logic, thus

[*] By size, we mean the number of object code characters it contains.

reducing the processing time for the search algorithm. These different binary encodings have various other properties which make them attractive. For example, one called "the Bϕ Method" allows variation codes to be created initially and never be modified as new operations are performed; another called the "CALL Method" encodes into the shortest binary strings, while the "LEVEL Method" is most closely related to the idealized stack viewpoint described above. Which one will provide the most useful is, as yet, undecided.

4. Programming Polyface Operations for Common Sub-Expressions

Variation encoding yields extreme complexity of data structure, and our initial attempt at programming TIE led to a three-page orgy of creating new beads, running around rings, and storing new pointer values. As TIE is one of the simplest of the eight Polyface operations, future prospects seemed gloomy. What was needed was a way of generating the Polyface operations from a lower level set of operations. One attempt was to define a simpler and more fundamental structure, called Polypoint, in which the atomic objects are points, and objects consist of sets of points. In this system the operations are few and uninteresting, but a new formation, called Polyline can evolve out of Polypoint. Certain non-atomic objects of Polypoint are redefined to be atomic lines in Polyline, and objects in Polyline are composed of these lines. A similar operation, with some additional side conditions then completes the evolution of Polyface from Polyline. Although in theory this two-stage evolution is possible, and this avenue may be pursued further in the future in its own right, the practical problems of programming the successive reinterpretations of the object beads for Polyface were formidable. For this reason a further alternate approach was sought, which would be more amenable to our present knowledge of plex programming.

The approach finally chosen was to seek some basic ring structure operations, which could then be used to program the Polyface operations. We hoped to accomplish this in the same way that the Generalized String Package was intended to be used in the original programming of Polyface. The primary difference is that the String

Package is concerned with operations on values with respect to abstract strings, whereas the new package is specialized to operations between pairs of rings. The same generalized implementation techniques (in terms of "basic functions" to describe detailed behavior) are used, and since the rings are instances of strings, the String Package functions can also be used, where appropriate.

The basic ring functions which we chose were designed to be a complete set of transformations between the elementary linked ring configurations as shown in Fig. 15. The basic ring functions are given the mnemonic names MERGE, JOIN, and KNOT. Their inverse

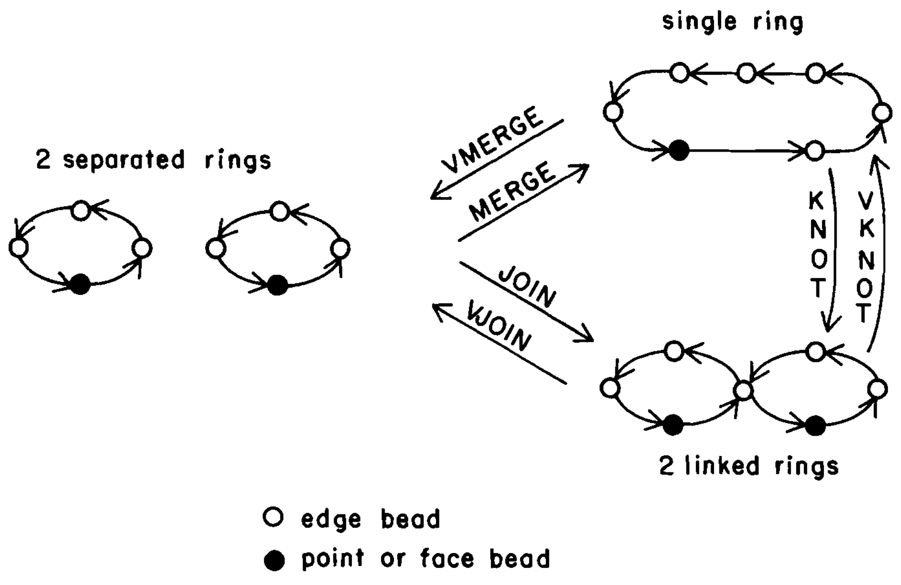


Fig. 15 The Six Basic Ring Functions

functions are given the same name with a "V" prefix. These ring functions have enabled us to realize the goal of reasonable program size for all of the Polyface Operations shown in Fig. 12.

D. SUMMARY

Research on the new Polyface Package for the modeling aspect of CADET has uncovered some new techniques which have general application within the framework of Plex theory. The system which was being programmed at the time work was terminated had the following characteristics:

1) The model is fully "common sub-expressioned", i.e., each distinct entity occurs only once in the system no matter how often it may be used.

2) The system keeps track of incremental changes only, by means of "variation beads", so that there is no redundancy.

3) The complete history of generation of a model is recoverable from the model.

4) Not only the structure of the total model (which may represent several alternate designs) is available, but also any substructure is uniquely isolatable at any time.

5) Generalized mouse algorithms have been devised which leave no "tracks" in the data structure of the model. That is, the complete state of the mouse is contained within itself so that any number of mice may be running simultaneously over the same model without interference.

6) The encoding of variations so that a mouse knows which of many variations to obey is done in a very compact binary code which uniquely identifies the precise location of a bead in the structure of the entire model.

Notice that making a variation ϕ_V of an object ϕ cannot change the appearance of other objects ϕ_i constructed from instances (calls) of ϕ prior to the creation of ϕ_V. This follows from characteristics 3) and 4) above, for if ϕ_i had been changed, the original form of these previous objects would have been lost, and we would be unable to model the entire history of the design sequence, or to isolate the ϕ_i substructure as they existed before ϕ_V was created.

In the final work on CADET under the contract, Polyface programs were partially debugged with a simple scaled display of polygons as a test vehicle. In order to provide a more suitable demonstration of the value of these techniques for the full CADET environment, it had been planned to include four specialized changes:

1) Change from 2-D to 3-D coordinates.

2) Allow nonplanar faces, nonlinear edges.

3) Incorporate Coons' surface patches as the face elements.

4) Elaborate the geometrical input language, to yield with the above a useful 3-D shape description facility.

Because of the termination of CADET activity during the final phase of the AED/360 bootstrap, none of these changes were actually made.

The basic theory behind Polyface also is far from finished. A generalized improvement would be to find a way to modify an object definition such that all calls in a specialized context include the new modifications, even though the calls themselves are generated previous to the change. This could easily be done with ordinary subroutine calls (merely by changing the definition) but then all calls would be effected, and there would be no control of context. On the other hand, when variations are used, the designer must reconstruct any objects which are to incorporate the new version if a basic element is changed. It appears, however, that augmentation of the basic Polyface operations by new operations to "edit" variation trees would allow changes to be made, with control over where these changes should apply. Care must be taken to preserve the desirable properties of the present compact variation code and mouse logic in any implementation of this new scheme.

Another very difficult but potentially valuable task is to design a "generalized Common Sub-Expression" package which would simplify the incorporation of these techniques in a wide spectrum of data modeling applications. The process of selecting out the common sub-expression portions of the Polyface package could also provide a start toward a "Generalized Ring" package similar to the String package. At present these two portions of the Polyface package are too entwined to be easily used as "Generalized" packages in other environments.

CHAPTER VI

COMPUTER GRAPHICS

The computer graphics efforts under Contracts F33615-67-C1530 and F33615-69-C-1431 during the reporting period were conducted by the ESL Display Group, which also received half of its support from Project MAC at M.I.T., which is funded by the Armed Forces Research Project Agency (ARPA) under Contract Nonr-4102(01). The following sections describe progress and important milestones during the reporting period in the computer buffering of the ESL Display Console constructed under previous contracts, in the development of a new low-cost graphic display (ARDS), and in graphics software for support of these display systems.

A. ESL DISPLAY CONSOLE

The ESL Display Console, which became operational in 1963, was the first graphics display system to incorporate such features as hardware rotation and scaling of displayed figures, features which have since become available in several commercial displays. From 1963 to 1967, the ESL Display Console operated directly from a data channel of the 7094 CTSS, utilizing the 7094 memory for display buffering. The display was completely described in the final report for previous contracts (Report AFML-TR-68-206), as well as plans for installation of a DEC PDP-7 computer to take over the buffering chores from the 7094.

The PDP-7 buffer computer for the original ESL Display Console became operational in September, 1967, shortly after the start of the reporting period. This greatly improved the performance of the display, as well as completely eliminating the former problem of the display using an excessive amount of 7094 computer time when it was directly driven by the 7094.

During 1966, a second ESL Display Console had been acquired by the M.I.T. Information Processing Center and operated briefly on a channel basis from their 7094 CTSS (which was a twin to the Project MAC CTSS). At the start of the reporting period, it was decided to move this console to the Electronic Systems Laboratory in Building 35, buffer it with a PDP-9 computer acquired by the ESL Computer-Aided Design Project under the previous Contract AF-33(657)-10954, and tie the PDP-9 to the Project MAC CTSS via a high-speed (50-kilobit) telephone link. Because the 7094 possessed no communications adapters for such data speeds, it was necessary to enter the Project MAC 7094 through the PDP-7, which already had channel-to-channel communication with the 7094. The main purposes of this tie-in were to experiment with communication procedures for remote buffered displays, and to provide a display capability in the Electronic Systems Laboratory. However, the configuration was planned with the future in mind. Since the PDP-7 and the PDP-9 would then be equipped for telephone-line communication, both displays could later be operated directly from MULTICS or the IPC 360-67 when these central time-shared facilities became equipped with suitable high-speed communication adapters.

The DEC Type 637 communications adapters for the PDP-7 and PDP-9 computers were installed in August, 1968, and the PDP-9/ESL Console interface was constructed and checked out. However, the Type 303 Datasets and connecting telephone line, which had been ordered from the New England Telephone Company in January, 1968, were not installed until May, 1969, seventeen months later! (This was due to the summer 1968 telephone strike and its aftermath.) As a result, no effective trials of remote display operation could be conducted under the contract, although much preparatory software work was done.

B. ARDS LOW-COST STORAGE-TUBE DISPLAY

The computer buffered refreshed displays discussed in the preceding section provide a dynamic graphics capability needed for certain types of problems, but the expense of such equipment precludes widespread installation. In order to provide the average time-sharing system user with a graphic capability, a low-cost display device is needed which can directly replace the existing teletype terminals. Starting in 1965, the Display Group began developing a new type of display terminal called ARDS (Advanced Remote Display Console) to provide high-speed alphanumerics and full graphical (picture-drawing) capability over an ordinary voice-grade telephone line.

By the start of the reporting period in May, 1967, the design and construction of prototype electronics for use with a direct-view storage tube were complete. The advantage of a direct-view storage tube for this application is that no local refresh memory is required, the character line generators need operate only fast enough to keep up with the incoming data on the telephone line, and more information can be displayed than on refreshed displays which are limited (by flicker considerations) in the amount they can display in one refresh cycle. The major remaining problem at the start of the present reporting period was that available storage tubes (five-inch) lacked adequate screen size and picture resolution. In August, 1967, however, a new storage-tube monitor became available; the Tektronix Type 611. This new tube has a screen size of 6-1/2 x 8-1/2 inches and has a stored spot size of 0.008 mils. Incorporation of this new tube with the prototype electronics immediately resulted in an outstanding alphanumeric and graphical display capability. A rather complete report of the ARDS development up to this point was given in Final Report AFML-TR-68-206 for the previous contracts.

During the latter part of 1967, the prototype electronics were further refined, including provisions for graphic input by means of an electronic cursor controlled by a joy-stick. A number of projects within M.I.T. and several organizations outside M.I.T. immediately became interested in acquiring ARDS, but the Project was unsuccessful in interesting existing display manufacturers in taking on the design. Thus in March, 1968, Messrs. R. H. Stotz and T. B. Cheek, principals in the development of the ARDS, left the Display Group and formed a company* to place the units on the market. Five units were ordered by Project MAC and other M.I.T. groups for connection to CTSS.

The first of these units was received in late June, 1968, and was immediately prepared for a rather unusual long-distance demonstration in conjunction with the joint M.I.T.- Technical University of Berlin

* Computer Displays, Inc.

Joint Summer Conference "Computers in the University," July 22 through August 3. A 1,200-bit link was set up via a dial-up connection from West Berlin to Frankfurt, Germany; ITT Datel Service from Frankfurt to New York; and a dedicated AT and T line from New York to the Project MAC computer. The ARDS was successfully operated for several hours each day of the conference. Figure 16 shows the table-top ARDS unit (center of the figure) in use at the Technical University in Berlin, with closed-circuit TV for audience viewing.

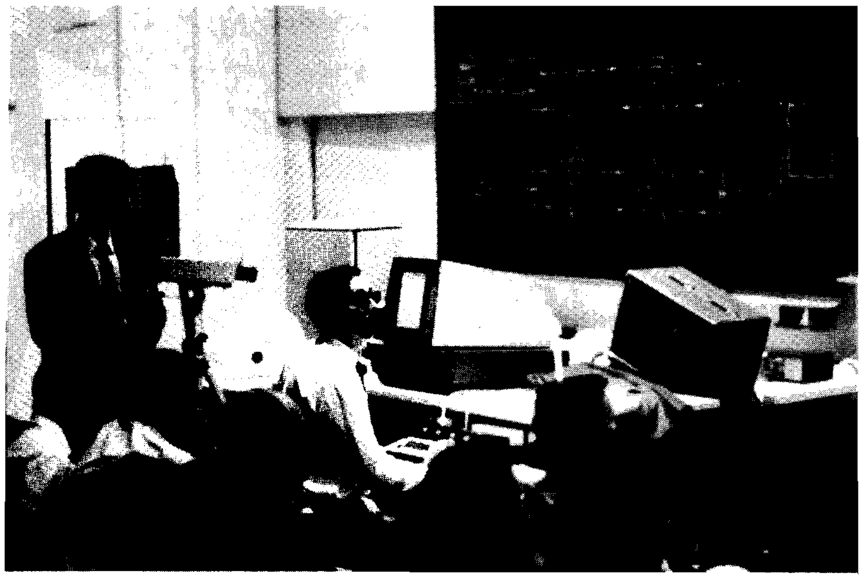


Fig. 16 ARDS Display Operation in Berlin, Germany, via Transatlantic Telephone Link

The 1200 bit-per-second transatlantic telephone link to the Project MAC 7094 computer at M.I.T. is shown on the blackboard in the background.

In January, 1969, the base price of the ARDS display was reduced to $8,000 (previously it had been $12,000), and additional units were ordered by various M.I.T. projects. Currently there are 19 ARDS in use at M.I.T., and about 100 others have been acquired by various industrial, academic, and governmental organizations.

A paper "Applying a Low-Cost Graphics Display" was presented by J. E. Ward at the 1969 IEEE International Convention in New York, March 24-27, 1969.

C. GRAPHIC SOFTWARE

1. Integration of the PDP-7 Buffer Computer

The initial programming of the Display Interface System for the buffered ESL Display Console was completed in September, 1967. The PDP-7 buffer computer now stores the display file, maintains the picture on the console, performs the real-time computations associated with control of the console hardware functions (rotation, translation, etc.), and processes display interrupts (light pen, push button, etc.) for users at two stations; functions which were previously performed by the IBM 7094 CTSS supervisor. This freed approximately 2500 registers of core storage in the supervisor, and reduced the load on CTSS for driving the display from the previous 3 to 20 percent to a negligible level. Accounting routines were written for the PDP-7 which enabled us to measure the portion of the PDP-7 time employed in maintaining the picture. It was found that only about 3 to 30 percent of the PDP-7's time is used for this purpose, depending on how heavy the real-time interaction is.

The 7094 programming packages for the ESL Display Console (Kludge) were originally called KLULIB. With the new modifications and additions made during the reporting period, the system has been renamed GRAPHSYS. The facilities available to the user in the PDP-7 console routines are the same as in the former 7094 routines, and at the GRAPHSYS (KLULIB) and A-core DSCOPE interfaces, the new buffered display interface system looks to the user exactly like the former unbuffered system. A complete description of the ESL Console, PDP-7 buffer system, and GRAPHSYS has been published in a joint ESL/Project MAC Report ESL-R-356/MAC-TR-56, "An Integrated Hardware-Software System for Computer Graphics in Time Sharing," December, 1968. The following is a summary of the essential features of the high-level programming interface provided by GRAPHSYS.

2. GRAPHSYS

GRAPHSYS is a set of procedures for programming interactive display consoles on time-sharing systems. As noted above, the system was originally written for the ESL Display Console connected to a time-shared 7094, but the user interface which it provides has also been used for programming the ARDS storage tube units. The following discussion pertains to the buffered refreshed-display situation (such as the ESL Console/PDP-7). The modifications for ARDS use will be discussed in Section C-3.

Throughout this discussion, the part of the time-sharing Supervisor and the executives of any satellite computers involved in running the display units will be called the Display Controller, or Controller, for short. The GRAPHSYS procedures, which provide a high-level language for programming the display console, reside in the "B-core" (non-supervisor) part of the time-sharing system along with the user's programs, and make calls on the Controller to effect the actions desired. This division of functions is shown in Fig. 17.

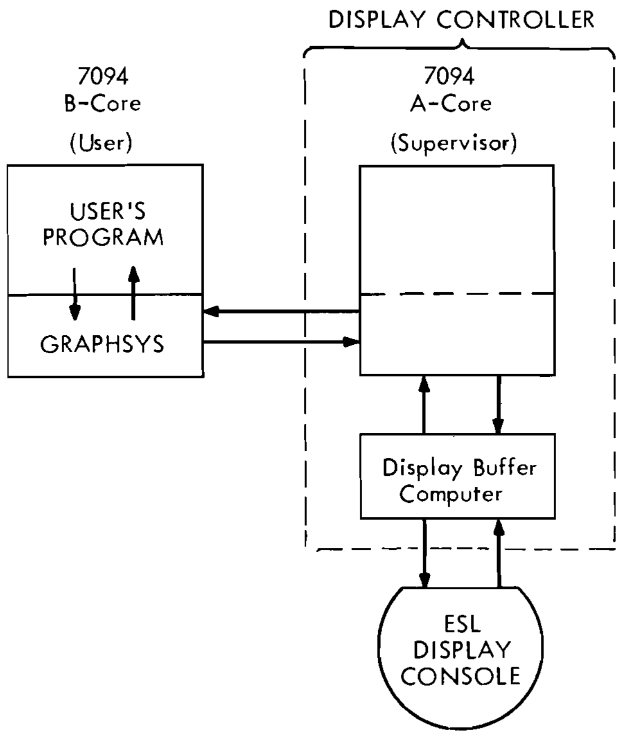


Fig. 17 Information Flow in Display System

The special problems associated with operating a display console in time-sharing, when the user has only intermittent access to the computer, have determined the division of tasks between GRAPHSYS and the Display Controller. The Controller performs the real-time functions: it stores the display file (an ordered sequence of display commands) and outputs them to the display unit to maintain the picture; it records attentions (real-time events such as pen "see") as messages for the user in an attention queue; and it performs certain real-time functions (e.g., rotation) as requested by the real-time instructions created by GRAPHSYS in response to user calls.

The Controller provides a single memory area called the display buffer for storing the display file, the attention queue, and the real-time instructions. The allocation and organization of this buffer is one of the tasks performed by GRAPHSYS. The user need only specify the number of registers to be used for the attention buffer (which contains the attention queue) and the real-time buffer (which contains the real-time instructions). The remaining space stores the display file proper, (and rotation matrix buffers, if the real-time rotation or magnification functions are used).

Flexible means are provided by GRAPHSYS for creating and editing a display file. Procedures for adding, removing, and replacing console commands, as well as a "copy" function whereby identical sets of commands may be reproduced within the display file, are included in the package. An important feature is the ability to define subroutine pictures or subpictures. Their definition is analogous to the way computer program subroutines are defined, and each time a display of a subpicture is desired, only a single command (the call command) is added to the display file. No provision is presently made for arguments to subpictures.

Each item added to the display file is assigned a name. All communication between the user and the system about items then takes place in terms of these names, the system automatically performing the required transformations for communication with the Display Controller. The names remain invariant even though the commands which they represent may be moved in the display file.

Thus GRAPHSYS provides a form of automatic storage allocation for the display file.

A set of procedures for adding standard picture parts to the display file is also provided. These include arcs of circles, lines, points, set-points, rotation matrix commands, control commands (affecting picture magnification, light pen sensitivity, etc.), and characters. A set point differs from a point in that its position remains fixed during rotations. A picture that is to be rotated consists of a single set point followed by a series of connected incremental vectors. A point is merely a unit length vector, (one scope increment in the plus direction followed by one in the negative direction so that the beam position is unaltered). The command which controls parameters such as intensity, sensitivity to the light pen, etc., is called a Set C command. The control command which increments the beam position directly, and is unaffected by rotation is called a Set F command.

The various facilities of the Display Controller may be called upon via GRAPHSYS. These include signing on and off with one or two consoles, placing the user's program in "input wait" when an attention is requested and there are none, reading the current values of the passive inputs, and requesting operation of the available real-time programs.

The GRAPHSYS procedures are written entirely in the AED-0 language, and use the AED-0 "Free Storage" System to allocate memory space dynamically for the B-core data structure. If the user employs free storage to store his own data structures in "plex" form, he may call the same free storage procedures, which are automatically loaded with GRAPHSYS. Since GRAPHSYS is written in AED-0, it may be bootstrapped to other computers. Such a bootstrap to the IBM 360 for use with ARDS displays was partially completed during the reporting period, and is currently in process of being completed by Project MAC.

3. <u>GRAPHSYS for ARDS</u>

The KLULIB programming system for the ESL Display Console was first modified in mid-1967 to provide a compatible programming interface to the ARDS, called ARDLIB. This package was revised and extended in a Master's Thesis research, completed in September, 1968, to add routines which allow the graphical input device (joy-stick, etc.) to be used like a light pen, and routines for defining "light buttons". Other software extensions include a display simulator package and provisions for storing ARDS pictures as CTSS disk files.

Changes and additions to GRAPHSYS were under consideration at the end of the reporting period, so that the programming interfaces for ARDS and the ESL Console can be made as near identical as possible. Work was also underway on planning for a whole new Display Interface System which would provide a single interface for the ARDS and the ESL Console (and perhaps other graphic devices); reduce the supervisor portion of the software to the bare minimum communications functions; and, in the case of the PDP-7/ESL Console, permit users to write their own real-time display routines. The general goal in this work is to make the whole graphics software system as display-independent and machine-independent as possible. Toward this end, ideas were being collected for a new user interface, and attention was being given to the problems of moving the whole system to new computers such as MULTICS and the 360-67.

A case-study example of the use of GRAPHSYS for programming ARDS was given as an appendix to Report ESL-R-356 cited previously. This example, using the drawing and editing of electrical circuits as a vehicle, was later greatly expanded under Project MAC support to illustrate more sophisticated aspects of interactive graphics programming, and published by Project MAC as a separate report:

"Case Study in Interactive Graphics Programming:
A Circuit Drawing and Editing Program
For Use with a Storage-Tube Display Terminal"

by

J. W. Brackett, M. Hammer, D. E. Thornhill

Report MAC-TR-63, Project MAC, M.I.T.
October, 1969, 94 pp.

CHAPTER VII

INDUSTRY PARTICIPATION

Since many Project activities broke new ground and required approaches quite different from those commonly used, a large amount of Project effort went into the preparation of technical presentations and documents describing the approach and the work. However, through past experience in the launching of the APT Language and System for programming of numerically-controlled machine tools, we early became convinced that we could not rely upon ordinary documentation and reporting to promote effective transfer of results into the application environment. Thus a more direct form of communication called "The AED Cooperative Program" was devised, as described in Section B.

In addition to the direct participation at M.I.T. by visiting industrial programmers under the AED Cooperative Program, many outside organizations received and worked with the several interim AED system releases (described in Chapter II) on their own computer facilities, and provided much valuable feedback. Usage of interim releases is detailed in Section C, and recipients of the final Version 3 AED/360 release since July, 1969 are given in Section D.

B. THE AED COOPERATIVE PROGRAM

Starting in early 1964, a number of industrial organizations were invited to participate directly in the work of the Project by assigning experienced system programmers to one-year visiting staff positions at M.I.T., to learn about and participate in on-going activities. It was felt that such cooperative liaison would be of value to both the individuals and organizations who accepted the invitation, and also provide important feedback from the intended application environment to help insure that the subsequent evolution of AED would be pertinent to the problems of industry. By active participation in the work itself, the industry representative at M.I.T. became skilled in the needed techniques being developed for the AED System, and learned many aspects which were impossible to document at the time. Furthermore, working programs developed in the Project were released immediately so that additional programmers in the company plants could also gain experience with the new techniques. Continual liaison was maintained between the M.I.T. Project and company personnel by direct communication, progress reports, and technical documentation, as well as meetings and visits.

Details of the first three years of the AED Cooperative Program, in which 18 companies participated, have been presented in the previous final report AFML-TR-68-206. During the present reporting period, there were ten visitors from nine companies. The complete participation over the five-year span of the AED Cooperative Program is shown in Fig. 18, and totals 362 man-months. Considering salaries and relocation expenses, this represents a total investment in AED by the participating organizations of the order of $1,000,000 to $1,500,000.

Complete names and affiliations of the AED visitors is shown in Table IV.

C. RECIPIENTS OF INTERIM AED SYSTEMS (1964 - 1968)

Starting in 1964, the M.I.T. Computer-Aided Design Project made available magnetic-tape copies of various interim-level AED systems to interested outside organizations. In each case, the recipient first sent a blank magnetic tape (or tapes) to M.I.T., and the only

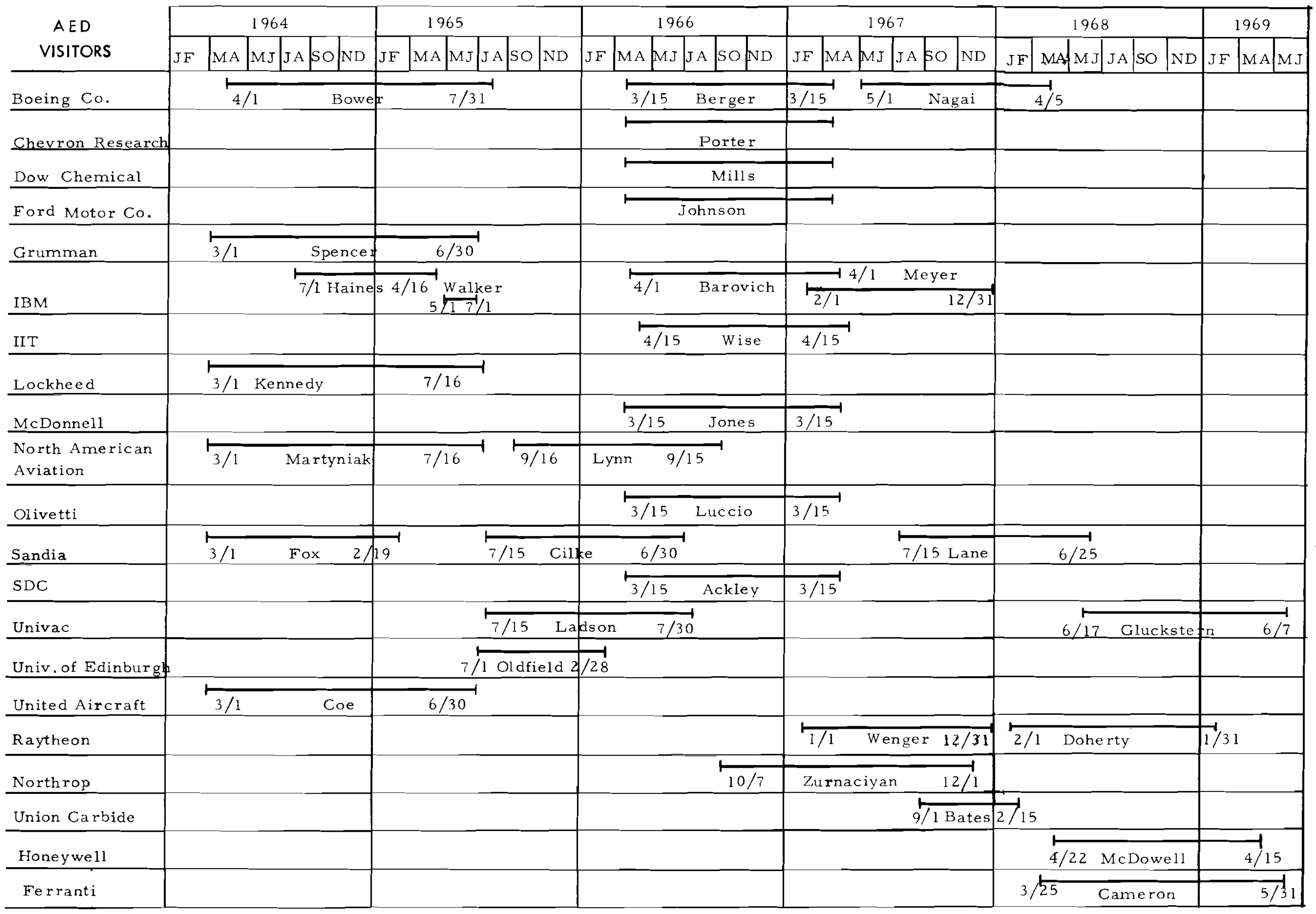


Fig. 18 AED Visiting Staff

Table IV

VISITING STAFF OF THE AED COOPERATIVE PROGRAM
(March 1, 1964 to June 30, 1970)

Stephanie I. Ackley	-	System Development Corporation
Donald Barovich	-	IBM Corporation
Frank Bates	-	Union Carbide Corp.
Anton J. Berger	-	The Boeing Company
Charles W. Bower	-	The Boeing Company
Donald J. Cameron	-	Ferranti, Ltd., Scotland
Howard J. Cilke	-	Sandia Corporation
Robert K. Coe	-	United Aircraft Corporation
John T. Doherty	-	Raytheon Manufacturing Co.
B. Thomas Fox	-	Sandia Corporation
Richard B. Gluckstern	-	Univac Div. of Sperry Rand Corp.
Leonard H. Haines	-	IBM Corporation
Walter L. Johnson	-	Ford Motor Company
Jack H. Jones	-	McDonnell Aircraft Corporation
James R. Kennedy	-	Lockheed-Georgia Company
Richard O. Ladson	-	Univac Div. of Sperry Rand Corp.
Fabrizio Luccio	-	Olivetti
Richard S. Lynn	-	North American Aviation
James J. Martyniak	-	North American Aviation
Robert J. McDowell	-	Minneapolis Honeywell EDP
Richard A. Meyer	-	IBM Corporation
Arthur K. Mills	-	The Dow Chemical Company
Arthur T. Nagai	-	The Boeing Company
John V. Oldfield	-	University of Edinburgh
James H. Porter	-	Chevron Research Company
Henry W. Spencer	-	Grumman Aircraft Corporation
June L. Walker	-	IBM Corporation
Irwin Wenger	-	Raytheon Company
Richard B. Wise	-	IIT Research Institute
Stephan Zurnaciyan	-	Northrup Corporation

expense to the contract was for the computer facilities (off-line) required for copying.

The systems released by M.I.T. were in three different categories: IBM 7094 versions, OS/360 versions, and 360/67 (CP/CMS) versions. The two 360 categories were either the Version 1 or Version 2 releases described in Chapter II. The OS/360 versions were for batch computing, the CP/CMS versions for time sharing. A fourth AED version for Univac 1108 computers was released by United Aircraft Corp., Hartford, Connecticut.

Table V lists all recipients of these various interim AED versions. Where more than one address is shown for a particular company, it indicates a separate release to a company division.

D. RECIPIENTS OF FINAL VERSION 3 AED/360 (1969 - 1970)

Version 3 AED/360 (described in Chapter III), was announced at the Third AED Technical Meeting held at M.I.T. on July 15, 1969, shortly before termination of the development group to form SofTech, Inc. Subsequently, arrangements were made for SofTech to provide an at-cost copying service similar to that previously provided by M.I.T. for previous versions. To date, 16 sets of tapes have been distributed, as shown in Table VI.

TABLE V

RECIPIENTS OF INTERIM AED SYSTEM RELEASES
1964-1968

IBM 7094 VERSION

BOEING COMPANY
Seattle, Wash.

CHEVRON RESEARCH
Richmond, Calif.

GRUMMAN AIRCRAFT
Bethpage, L.I.

HUGHES AIRCRAFT
Culver City, Calif.

IIT RESEARCH INSTITUTE
Chicago, Ill.

IMPERIAL COLLEGE
London, England

IBM CORP.
Marietta, Ga.
Poughkeepsie, N.Y.

LOCKHEED GEORGIA
Marietta, Ga.

McDONNELL DOUGLAS CORP.
Huntington Beach, Calif.
St. Louis, Mo.

MOBIL OIL CORP.
New York, N.Y.

NORTH AMERICAN ROCKWELL
Downey, Calif.
El Segundo, Calif.

N.V. PHILIPS G.
Eindhoven, Netherlands

SANDIA CORP.
Albuquerque, N. M.

UNION CARBIDE
Oak Ridge, Tenn.

UNITED AIRCRAFT
East Hartford, Conn.

U.S. NAVY DEPARTMENT
Washington, D.C.

OS/360 VERSION

BOEING COMPANY
Seattle, Wash.
Wichita, Kansas
New Orleans, La.

BELL HELICOPTER CORP.
Fort Worth, Texas

CHEVRON RESEARCH
Richmond, Calif.

GRUMMAN AIRCRAFT
Bethpage, L.I.

IBM FEDERAL SYSTEMS DIV.
Rockville, Md.

LITTON SYSTEMS INC.
Woodland Hills, Calif.

MOBIL OIL COMPANY
New York, N.Y.

NORTH AMERICAN ROCKWELL
Downey, Calif.

NORTHROP CORP.
Hawthorne, Calif.

PENN. STATE UNIVERSITY
University Park, Pa.

SANDERS ASSOCIATES
Nashua, N.H.

TRW SYSTEMS
Redondo Beach, Calif.

UNITED AIRCRAFT CORP.
East Hartford, Conn.

U.S. NAVAL WEAPONS LAB.
Dahlgren, Va.

ABCOR, INC.
Cambridge, Mass.

IMPERIAL COLLEGE
London, England

N.V. PHILIPS G.
Eindhoven, Netherlands

UNIVERSITY COLLEGE
London, England

360/67 (CP/CMS) VERSION

MASS. INST. OF TECHNOLOGY
Lincoln Laboratory
Urban Systems Laboratory

IBM CORP.
Cambridge Scientific Center

UNIVAC 1108 VERSION

BOEING COMPANY
Seattle, Wash.

COMPUTER SCIENCES CORP.
Huntsville, Ala.

JET PROPULSION LABORATORY
Sunnyvale, Calif.

NATIONAL ENGINEERING LABORATORIES
Glasgow, Scotland

UNITED AIRCRAFT CORP.
East Hartford, Conn.

UNIVAC
Cambridge, Mass.

TABLE VI

RECIPIENTS OF FINAL VERSION 3 AED/360 (1969-1970)

	OS	CP/67
BENDIX INDUSTRIAL CONTROLS Detroit, Michigan	*	
THE BOEING COMPANY Seattle, Washington	*	
BROWN UNIVERSITY Center for Computer and Info. Sci., Providence, Rhode Island		*
CODON COMPUTER UTILITIES Waltham, Massachusetts	*	*
COMPUTER SOFTWARE SYSTEMS, INC. Stamford, Connecticut		*
GRUMMAN AIRCRAFT ENG. CORP. Bethpage, L.I., New York	*	
IIT RESEARCH INSTITUTE Chicago, Illinois	*	
INTERACTIVE DATA CORP. Waltham, Massachusetts		*
IBM CAMBRIDGE SCIENTIFIC CENTER Cambridge, Massachusetts		*
MASSACHUSETTS INSTITUTE OF TECHNOLOGY Cambridge, Massachusetts	*	*
NATIONAL SECURITY AGENCY Ft. George C. Meade, Maryland	*	
PENNSYLVANIA STATE UNIVERSITY AE Computer Aided Design Lab., University Park, Pennsylvania	*	*
PHILIP HANKINS, INC. Arlington, Massachusetts	*	
PHILIPS RESEARCH LABS. Eindhoven, Holland	*	
U.S. NAVAL WEAPONS LABORATORY Dahlgren, Virginia	*	
WESTINGHOUSE ELECTRIC CORP. Information Systems Division, Pittsburgh, Pennsylvania	*	

CHAPTER VIII

ABSTRACTS OF PROJECT PUBLICATIONS

The content of the various technical reports published since May, 1968, has been summarized and referenced in previous chapters. For reference convenience, the report titles and abstracts are all listed below in chronological order of publication.

Since this was a continuing effort, based on prior work going back a number of years, two previous final reports summarizing the prior work are also listed to provide a ready reference to prior reporting.

A. FINAL REPORTS FOR PREVIOUS CONTRACT PERIODS

1. Final Report 6873-FR-3, "Automatic Programming of Numerically Controlled Machine Tools," John E. Ward, January 15, 1960, 120 pp.

   This report covers the three-year period of development work on the APT System which preceded the start of active work on computer-aided design in 1959. It documents the first discussions of "computer aids to design", carried out as one task during the final year of the APT work. It also contains reprints of Final Reports 6873-FR-1 and 6873-FR-2 on the development of numerical control, thus serving as a reference to all N/C MIT work of the Air Force under Contract AF-33(038)-24007 from February, 1951 through November 30, 1959.

2. Report AFML-TR-68-206 (ESL-FR-351), "Investigations in Computer-Aided Design for Numerically Controlled Production," D. T. Ross and J. E. Ward, May 1968, 243 pp.

   This report summarizes the activities of the M.I.T. Computer-Aided Design Project from 1 December 1959 through 3 May 1967 in the development of a generalized "system of software systems" for generating specialized problem-oriented man-machine problem-solving systems using high-level language techniques and advanced computer graphics. Known as the AED Approach (for Automated Engineering Design) the Project results are applicable not only to mechanical design, as an extension of earlier development of the APT System for numerical control, but to arbitrary scientific, engineering, management, and production system problems as well. All results have been programmed using machine-independent techniques in the Project's AED-0 Language, based on Algol-60, and are operational on several widely available computers.

Advanced techniques for verbal and graphical language and generalized problem-modeling are based on the concept of a plex which combines data, structure, and algorithmic aspects to provide complete and elegant representation of arbitrary problems. Program developments are supported by hardware and software innovations in computer graphics. Various design applications and a general technique for three-dimensional shape description complement and illustrate the general approach. The unique AED Cooperative Program allows visiting staff from industry to learn Project results while contributing to their further development.

A complete bibliography of 274 references to Project documents, talks, and thesis reports is included.

B. TECHNICAL REPORTS FOR PRESENT REPORTING PERIOD

3. Report ESL-R-356 (MAC-TR-56), "An Integrated Hardware-Software System for Computer-Graphics in Time-Sharing," D. E. Thornhill, R. H. Stotz, D. T. Ross, and J. E. Ward, December 1968, 168 pp.

This report describes the ESL Display Console and its associated user-oriented software systems developed by the M.I.T. Computer-Aided Design Project with Project MAC. Console facilities include hardware projection of three-dimensional line drawings, automatic light pen tracking, and a flexible set of knob, switch, and push-button inputs. The console is attached to the Project MAC IBM 7094 Compatible Time-Sharing System either directly or through a PDP-7 Computer. Programs of the Display Controller software provide the real-time actions essential to running the display, and communication with the time-sharing supervisor. A companion graphics software system (GRAPHSYS) provides a convenient, high-level, and nearly display-independent interface between the user and the Display Controller. GRAPHSYS procedures allow the user to work with element "picture parts" as well as "subpictures" to which "names" are assigned for identification between user and Controller programs. Software is written mostly in the machine-independent AED-0 Language of the Project and many of the techniques described are applicable in other contexts.

4. Technical Memorandum ESL-TM-394, "The SHOWIT System: An Example of the Use of the AED Approach," J. R. Ross and D. T. Ross, June 1969, 98 pp.

The AED approach is a collection of programming concepts and working tools for use in creating specialized, problem-solving systems. This report is a tutorial document discussing an example of the application of the AED approach, with particular emphasis on the facilities of the AEDJR parsing processor for implementing new languages. The report is largely self-contained and should be useful to both experienced and novice AED users. The report is in two parts; Part One contains a summary of the AED approach and the manner of using it in practical situations. Part Two contains a detailed description of SHOWIT, the tutorial example system. Listings of the SHOWIT program files, heavily interspersed with comments, are included in the Appendix.

5. Report ESL-R-395 (MAC-TR-62), "EPS: An Interactive System for Solving Elliptic Boundary-Value Problems with Facilities for Data Manipulation and General-Purpose Computation -- USER'S GUIDE," C. C. Tillman, Jr., June 1969, 187 pp.

This appendix for the author's forthcoming thesis, "On-Line Solution of Elliptic Boundary-Value Problems," is a user's guide for EPS. EPS solves two-dimensional boundary-value problems for elliptic systems of second-order partial differential equations. It also has general-purpose capabilities which permit the on-line definition and execution or arbitrary numerical procedures.

The guide is concerned primarily with the use of EPS for solving elliptic boundary-value problems. Linear problems of this type that have no complications such as free surfaces or undefined parameters can be solved on a one-pass basis. Nonlinearities and other complications can be accommodated by iteration. Solutions are obtained by a finite-difference method which permits the use of irregular lattices, hence the crowding of nodes in sensitive regions.

EPS operates on the IBM 7094 computer of the M.I.T. Compatible Time-Sharing System (CTSS), and exploits to an unusual degree the potential for interactive problem solving that CTSS affords. Input commands resemble statements in various algebraic compiler languages, and can be combined and abbreviated by means of macros. Improper input and other error conditions are handled so as to minimize user inconvenience. Common syntax errors, for example, are corrected automatically by the machine. Output is available in either numerical or graphical form.

6. Report ESL-R-397, "An Interactive Syntax Definition Facility," R. S. Eanes, September 1969, (Also SM Thesis in Department of Electrical Engineering, M.I.T.), 69 pp.

The Syntax Definition Facility (SDF) is an interactive system which allows the designer of a computer programming language to define the syntax of his language in a relatively simple natural meta-language. From this syntax definition a set of tables for driving a general parsing algorithm are produced. If the system detects possible ambiguities or inconsistencies in the definition supplied by the language designer, it will report them and try to indicate the source of the problem. The system includes test and debugging facilities to aid the language designer.

The SDF meta-language allows the language designer to specify the syntax of his language by writing a series of sample statements which are marked to indicate how they should be parsed. Each sample statement specifies the "kind of value" or semantic type of a construction in the language. By the underlying principle of phrase substitution, which allows any construction to be substituted for any other construction of the same semantic type, the small number of sample statements induces a complete language definition allowing statements of arbitrary size and complexity. A syntax definition in the SDF meta-language is somewhat similar to a Backus-Naur Form or context-free grammar definition, but it is more readable and easier to produce.

The algorithm used by the system to produce a parser from the meta-language description is a synthesis of the precedence techniques and the AED language definition systems.

7. Report ESL-R-398 (MAC-TR-64), "A Graph Model for Parallel Computations," J. E. Rodriguez, September 1969, (Also ScD Thesis in Department of Electrical Engineering, M.I.T.), 130 pp.

This report presents a computational model called program graphs which makes possible a precise description of parallel computations of arbitrary complexity on non-structured data. In the model, the computation steps are represented by the nodes of a directed graph whose links represent the elements of storage and transmission of data and/or control information. The activation of the computation represented by a node depends only on the control information residing in each of the links incident into and out of the node. At any given time any number of nodes may be active, and there are no assumptions in the model regarding either the length of time required to perform the computation represented by a node or the length of time required to transmit data or control information from one node to another. Data dependent decisions are incorporated in the model in a novel way which makes a sharp distinction between the local sequencing requirements arising from the data dependency of the computation steps and the global sequencing requirements determined by the logical structure of the algorithm.

The concept of the state of a program graph is introduced and it is proved that every program graph represents a deterministic computation, i.e., that the final state of each computation started from the same initial state is unique. Computations which do not terminate properly are defined in terms of the concept of hang-up state. Methods of analysis are developed and necessary and sufficient conditions for the absence of hang-up states are obtained. These conditions are interpreted in terms of the structure of the graph and the manner in which the decision elements are imbedded in that structure. Finally, an equivalence problem for program graphs is formulated and a solution to this problem is presented.

8. Technical Report MAC-TR-63, "Case Study in Interactive Graphics Programming: A Circuit Drawing and Editing Program for Use With a Storage-Tube Display Terminal," J. W. Brackett, M. Hammer, and D. E. Thornhill, October 1969, 100 pp.

The concepts involved in building and manipulating a data structure through graphical interaction are presented, using the drawing and editing of electrical circuits as a vehicle. The circuit drawing program was designed to operate on an ARDS storage-tube display terminal attached to the M.I.T. Project MAC IBM 7094 Compatible Time-Sharing System. The graphics software system (GRAPHSYS) developed by the M.I.T. Computer-Aided Design Project was used for dealing with all graphical input and output, and the AED Language of the Project was used in programming. AED System packages for building and manipulating complex data structures are described and their use is illustrated in detail. The report includes flow diagrams and complete listings of the sample circuit drawing and editing system.

9. Report ESL-R-405, "Introduction to Software Engineering with the AED-0 Language," D. T. Ross, October 1969, 241 pp.

This report is a tutorial exposition of the AED-0 language and of a software engineering discipline based upon its use. The AED (Automated Engineering Design) System has been developed over a ten-year period by the M.I.T. Computer-Aided Design Project, culminating in release of the Version 3 AED-1 Compiler and associated system-building systems for use on IBM 360-series computers in both batch and time-sharing. Bootstraps to other computers are in progress.

The AED-0 language, based on ALGOL-60, is the present language for the AED-1 Compiler. The report describes a major subset of the AED-0 language and demonstrates its use. A companion report, the AED Programmer's Guide, specifically covers the Version 3 release, and presents additional features of the AED-0 language.

(See Table II, page 53, for complete Table of Contents.)

10. Report ESL-R-406, "AED-0 Programmer's Guide," C. G. Feldmann, D. T. Ross, and J. E. Rodriguez, January 1970, 336 pp.

The AED (Automated Engineering Design) System has been developed over a 10-year period by the M.I.T. Computer-Aided

Design Project. This report is a user reference manual for the AED-0 language and the Version 3 AED-1 compiler and associated system building packages, as released in July, 1969 for use on IBM 360-series computers in both batch-oriented and time-sharing operating systems. Part 1 of the manual describes the AED-0 language proper, including descriptions of several subroutine packages which extend the features of the language beyond the forms derived from Algol-60 syntax. Part 2 describes additional subroutine packages useful as building blocks for general software construction.

The material is organized so that a given aspect of the AED-0 language or subroutine library is discussed in complete detail, from basic to advanced features, in a separate chapter. A tutorial approach to the AED-0 language and its use is given in a companion report, ESL-R-405, Introduction to Software Engineering with the AED-0 Language, and is recommended for collateral reading.

Bootstraps of AED to other computers were partially completed under the contract, and are being completed outside M.I.T. This report will also serve as a user reference manual for these versions.

(See Table III, page 60, for complete Table of Contents.)

APPENDIX A

AED QUESTIONNAIRE

Simultaneously with the 9 June 1969 announcement of the Version 3 AED/360 release and the 15 July 1969 Third AED Technical Meeting, the following questionnaire was sent to all past recipients of AED systems, with a requested return date of 27 June 1969. Results of the 54 questionnaires returned are summarized in Chapter II (Section G).

The AED (Automated Engineering Design) programs and systems have been reworked and re-bootstrapped during the present, terminal Air Force contract with the M.I.T. Electronic Systems Laboratory to produce fully releasable versions of the following programs for IBM System 360 computers in both batch-processing and time-sharing environments:

1) The AED-1 Compiler for the AED-0 Language
2) The AEDJR System for language definition
3) The AED Library of system-building packages

This release, plus progress toward a fully compatible release for the Univac 1108 will be described at the Third AED Technical Meeting to be held at the M.I.T. Kresge Auditorium on Thursday, July 15, 1969. Also to be discussed are new formal documentation which has been prepared on the use of the AED Language and systems, AED applications, and formation of a User's Group. In connection with preparations for this meeting, it would be most helpful if you could take a few moments to answer the following questions:

Name: Position:

Company:

Address:

1.) Have you studied AED in your organization? Yes ____ No ____
If Yes, what do you think are the strongest and weakest features of AED?

Strongest:

Weakest:

2.) Has AED been used in your organization? Yes ___ No ___
If No, why not?

If Yes, which pre-release systems have been used?

7094 ___ 1108 Exec2 ___ 360 OS ___ 360 CP/CMS ___

How many people have learned AED-0?

Engineers ____ Programmers ____ System Programmers ____

How many projects have used AED? ____
(Question 6 requests brief descriptions.)

3.) AED has many areas of potential applicability. Please compare with other programming languages for the following categories of use.

	Research Investigations	General Programming	System Building
AED			
FORTRAN			
COBOL			
PL/1			
Assembly			
Low	0 1 2 3 4 5 High	0 1 2 3 4 5	0 1 2 3 4 5

	Computer Graphics	Data Base and Management Systems	Other
AED			
FORTRAN			
COBOL			
PL/1			
Assembly			
Low	0 1 2 3 4 5 High	0 1 2 3 4 5	0 1 2 3 4 5

4.) Various users of a system have different needs with respect to system support. Please rate your interest in the following support services if you are using or planning to use AED:

Improved User Documentation	
System Maintenance	
Training Courses	
Direct Use Assistance	
Low	0 1 2 3 4 5 High

5.) Assuming that at least the first two of the items of support listed in Question 4 were available, please indicate expected use of AED in your future work. (Please make a mark for each year indicated.)

	Research Investigations	General Programming	System Building
1973			
1972			
1971			
1970			
Present			
Low	0 1 2 3 4 5	High 0 1 2 3 4 5	0 1 2 3 4 5

	Computer Graphics	Data Base and Management Systems	Other ______
1973			
1972			
1971			
1970			
Present			
Low	0 1 2 3 4 5	High 0 1 2 3 4 5	0 1 2 3 4 5

6.) Brief descriptions of any projects for which AED has been used or has been considered in your organization would be of value. We would appreciate your attaching descriptions of such projects, including the overall effectiveness of AED for each project.

APPENDIX B

(PAPER REPRINT)

Automatic Generation of Efficient Lexical Processors Using Finite State Techniques

WALTER L. JOHNSON, *Ford Motor Company, Dearborn, Michigan*
JAMES H. PORTER,* *Chevron Research Company, Richmond, California*
STEPHANIE I. ACKLEY, *System Development Corporation, Santa Monica, California*
DOUGLAS T. ROSS, *Massachusetts Institute of Technology, Cambridge, Massachusetts*

The practical application of the theory of finite-state automata to automatically generate lexical processors is dealt with in this tutorial article by the use of the AED RWORD system, developed at M.I.T. as part of the AED-1 system. This system accepts as input descriptions of the multicharacter items or of words allowable in a language given in terms of a subset of regular expressions. The output of the system is a lexical processor which reads a string of characters and combines them into the items as defined by the regular expressions. Each output item is identified by a code number together with a pointer to a block of storage containing the characters and character count in the item.

The processors produced by the system are based on finite-state machines. Each state of a "machine" corresponds to a unique condition in the lexical processing of a character string. At each state a character is read, and the machine changes to a new state. At each transition appropriate actions are taken based on the particular character read.

The system has been in operation since 1966, and processors generated have compared favorably in speed to carefully hand-coded programs to accomplish the same task. Lexical processors for AED-0 and MAD are among the many which have been produced.

The techniques employed are independent of the nature of the items being evaluated. If the word "events" is substituted for character string, these processors may be described as generalized decision-making mechanisms based upon an ordered sequence of events. This allows the system to be used in a range of applications outside the area of lexical processing.

However convenient these advantages may be, speed is the most important consideration. In designing a system for automatic generation of a lexical processor, the goal was a processor which completely eliminated backup or rereading, which was nearly as fast as hand-coded processors, which would analyze the language and detect errors, and which would be convenient and easy to use.

KEY WORDS AND PHRASES: character string, compiler, finite-state automata, finite-state machine, lexical processor, nondeterministic machine, parsing, plex structure, regular expressions, sequential machine, syntactic analysis
CR CATEGORIES: 3.63, 3.75, 4.12, 5.22, 5.24, 5.31

Introduction

The functioning of a compiler can be described in the following four logically separable steps:

1. Read the characters of the source language and assemble them into meaningful words or items (lexical analysis);
2. Group the "words" of the language into phrases and sentences for analysis (parsing);
3. Extract the meaning of the source language sentences (modeling);
4. Produce the appropriate object code.

Indeed, if step 4 above is changed to read "Carry out the solution to the problem posed by the source language" and generalize "characters" to mean communication signal elements, our four steps now apply to computer solution of problems in general [1].

Step 4, in the general case, represents the algorithm for the particular problem and as such is not amenable to general solutions. In the case of compilers some work has been done in this area [1, 6].

Step 3, the analysis of the parsed language, again is not amenable to general solutions, although some work has been done toward providing a general mechanism for this analysis [3].

Step 2 is much more amenable to generalized solution. Very general systems exist for producing special-purpose processors for parsing language [2–5]. Such automatically produced parsers are being used in compilers and other computer problem-solving applications.

With some exceptions, lexical properties have been assigned a very minor role in computer languages, and lexical processing has been incorporated as an incidental

The work reported here was made possible through support and sponsorship extended to the MIT Electronic Systems Laboratory by the Manufacturing Technology Laboratory, RTD, Wright-Patterson Air Force Base, under Contract F 33 615-67-C-1530, MIT Project DSR 70429. The work was also supported (in part) by Project MAC, an MIT Research Program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract NOnr-4102(01).

*Present address: Abcor, Inc., Cambridge, Mass.

part of the syntactic-analysis programs. There are some good reasons for separating these functions both logically and programmatically [5].

1. A large portion of compiler time is consumed in lexical analysis, making it essential that this function be as efficient (fast) as possible. Conway [4] notes that in his experience with a prototype compiler the input speed difference between lexical analysis alone and lexical analysis plus syntactical analysis was about 10 percent. Separating out the lexical analysis allows this problem to be attacked more effectively.

2. The development of effective languages requires attention to the lexical as well as the syntactic properties of the languages. Separating the two functions promotes recognition of this fact, and allows the functions to be investigated independently.

3. Separation allows the development of systems for automatic syntactic and lexical analysis.

The third point is particularly important, since the existence of such systems allows the language developer or compiler writer to experiment with various lexical and/or syntactic schemes without the burden of the immense programming times which would otherwise be required.

Several such systems already exist for generation of syntactic processors [11]. In this paper a system is discussed for the generation of lexical processors which was developed at M.I.T. as part of the AED-1 system [1], while the three non-M.I.T. authors were representatives of their respective companies in the AED Cooperative Program of the M.I.T. Computer-Aided Design Project, in 1966–67.

The RWORD System

As described in [1], the AED approach to the generation of man/machine problem-solving systems involves the use of several auxiliary systems to prepare processors for the major phases of a final specialized system. The system for producing lexical processors is referred to as the Rword (read a word) system. The system has the following general properties:

Input. The user specifies the lexical properties of his language to the Rword system by specifying the makeup of the items (words) which are allowable in the language. This is done by means of a limited subset of the language of regular expressions [7, 8] which is used to describe how individual characters and/or character classes are combined to form items. The format by which the user describes an item to Rword is as follows:

item name (item code) = regular expression $,

where

item name—is a mnemonic name for the lexical type of item being described such as INTEGER and VARIABLE.

item code is an integer number which provides an "internal code" for the lexical type. It is used to identify the item type to programs when an item is encountered in the input stream.

regular expression Table I describes the operators which can be applied to individual characters or user-defined character classes such as LETTER and DIGIT to form expressions describing the composition rules for items of a lexical type.

$, a terminator which indicates the end of the regular expression and of the item description.

If character classes are referred to, the individual characters in each class must also be specified to the Rword system. Table II shows an example of Rword input.

Output. The output of Rword is a table of information which controls the operation of a system-supplied run-time package of lexical processing procedures. The combination of the table and procedures is the special lexical processor which will itemize a character string in the manner specified by the input to Rword. The lexical processor is activated from a compiler or other user

TABLE I. Regular Expression Operators Used in RWORD Input

Operator	*Meaning*	*Example*	*Interpretation*
/	concatenate	A/B	A concatenated with B, i.e. AB
U	union, or	A U B	A or B
*	none or more of the preceding	B/A*	B concatenated with none or more A's, i.e. BAA or B or BAAA etc.
*n	none or more, but not exceeding n	B/A*2	B or BA or BAA
( )	the usual phrase separation	A/(B U C)	A concatenated with B or C, i.e. AB or AC
=	is defined as	EXAMPLE (1) = A U B $,	The item type named "EXAMPLE" with internal code "1" is defined as A or B
$,	item-description terminator		
'	the following character is not to be treated as an operator	'/ /A	/ followed by A
NULL	no character	A/(B U NULL)	AB or A

Note. Parentheses on the left of the equal sign (after the item name) enclose the item type code number to be used by RWORD's output processor to identify the lexical type of the item. The name of the "execute" procedure, if any, to be called by the processor on encountering this item type may also be included: EXAMPLE (1, ACTION) = ...

system through a call on one of the procedures in the package called "NX.ITM". NX.ITM returns one item on each call, setting the item code, and setting a pointer to a block of storage words containing the internal coded representation of the characters of the item, and a character count. Figure 4 depicts the construction of the RWORD system.

Because a large portion of compilation time is usually consumed in lexical analysis, one of the prime considerations in the design of the RWORD system was the efficiency of the lexical processors. Although it is impossible to separate out the effects of different operating system environments, in limited tests to date, RWORD's processors have compared favorably with carefully hand-coded FAP processors performing similar functions. On the IBM 7094 a carefully hand-coded lexical processor using highly specialized techniques for the AED-0 language processed characters at a rate of 8000 characters per second. An RWORD-generated processor using an extremely flexible implementation accomplished the same task at a rate of 3000 characters per second. New implementation techniques which isolate special actions and thereby permit the high frequency "normal" actions to take place with minimized overhead are expected to significantly reduce this difference while still maintaining the generality and ease-of-use of the overall scheme.

Context Dependence

Many computer languages require context-dependent lexical processing, a simple example being the special treatment accorded characters within remarks or comments.

Context dependence is handled in RWORD by constructing a separate processor for each context situation and switching between these processors when the appropriate characters or items are encountered. This is accomplished using the "execute" functions described below.

Nonlexical Functions

The RWORD system also allows the user to specify as input names of "execute" procedures which are to be called by the processor whenever certain items or character classes are encountered. This is done by including the

TABLE II. EXAMPLE OF RWORD INPUT

Class descriptions	*Meaning*	*Class descriptions*	*Meaning*
BEGIN	The first input line must be BEGIN	SYM(1) = LET/(LET ∪ DIG)*5$,	The item type to be identified by the item code "1" is to consist of a character of the class "LET" (letter) followed by zero-to-five characters of the classes "LET" or "DIG" (digit).
LET = ,ABCDEFGHIJKLMNOPQRSTUVWXYZ/	The class LET consists of all the letters. The first nonblank character after the equal sign is treated as a delimiter.		
DIG = ,1234567890/		INTEGER(2) = DIG/DIG*$,	The item type of code number "2" is to consist of one-or-more characters of the class "DIG".
SPACE = / /			
IGNORE = $12, 15, 16, 17, 32, 35, 36, 37, 52, 55, 56, 57, 72, 75, 76$	When a class is named "IGNORE" the resulting processor "reads over" all characters of that class when processing the input string. The characters are defined by the internal coded representation of characters on the machine being used. Each character is separated by a comma. When the dollar sign ($) is used as a delimiter RWORD treats what follows as two-digit BCD codes for the characters.	FRACTION(3) = ./DIG/DIG*$,	Item type "3" is to be a dot followed by one-or-more DIG's.
		IGNORE = SPACE $,	RWORD allows designation of ignorable items in the manner shown. Here space, being an item in itself, will act as a delimiter, but will not be "reported" as an item by the processor.
		REMARK(4,SWITCH1) = ././.$,	Item 4 is three dots. The procedure SWITCH1 will be called by the processor whenever item 4 is encountered. The procedure SWITCH1, is called *before* the item is reported by the processor.
PUN = A.,+−=()$/A	Here "A" acts as the delimiter.		
END	Class descriptions terminate with END	PUNCT(5) = PUN $,	Item 5 is any character of the class PUN.
BEGIN	The first input line must be BEGIN	END FINI	The item descriptions terminate with END FINI.

procedure name along with the particular item description or character class description. The user himself writes these procedures to match standardized calling contexts of the package routines and loads them with the table and the run-time package. The resulting processor then stops at the designated spots in reading the character stream and calls the user-supplied program. An execute program may switch lexical processing schemes, may trigger the next level of language analysis (as in interpretive systems when a statement terminator is encountered), or may perform any other function that the nature of the application dictates.

Simple Languages

Although Rword can handle very complex lexical problems, there exists a large and useful subset of lexical schemes (which we call "simple languages") which can be handled by a simpler processing mechanism than that required in the general case. Simple languages are ones in which, upon reading each character, the processor can determine unambiguously if that character starts a new item or if it should be included as part of the same item as the previous character.

Fortran IV illustrates a complex language. Consider the following character strings:

Strings	*Item structure*	*Number of items*
2.EQ.K	2 .EQ. K	3
2.E10	2.E10	1

A simple processor does not allow these two strings, since it is necessary to read past the "E" before the disposition of the "." can be determined. A nonsimple processor must cope with this type of look-ahead. A simple processor makes a decision before it reads the next character.

The Rword system detects complexities in languages and gives the user the most efficient processor his description allows. The user also has the option of treating his inherently complex language with a "simple" processor, in which case the processor always elects to add a new character to the previous item when it may. This choice has been found to satisfy the intent of most languages which have been tried to date. In the example above, it would correspond to the convention that the first string must have a space if an alarm is to be avoided. This seems to be a generally acceptable kind of convention since it also improves the readability of programs.

Ambiguous Languages

Frequently, sets of item descriptions input to Rword allow a given character string to be itemized in more than one way. Such languages are said to be ambiguous. Ambiguities are often very difficult to detect from inspection of the regular expressions describing the items. Rword detects all such ambiguities and takes one of three actions.

1. Makes a decision resolving the ambiguity, and proceeds. This is done in such cases as a Fortran variable name (one-to-six letters or digits, starting with a letter) in which the character string AB could be grouped as "A" and "B" or "AB", both of which satisfy the same definition. Rword makes a processor which takes the longest item in such cases, and proceeds without reporting the situation.

2. Reports the ambiguity, makes a decision, and proceeds. Certain ambiguities have less clear-cut resolutions than the above, but Rword makes an educated guess as to the user's intention and proceeds. *Example:* The characters X123 could be grouped as the variable name X followed by the integer 123 or as the variable name X123. Since two separate definitions are involved, Rword reports a possible ambiguity, elects to make a processor which will make the longest item in such cases, and proceeds.

3. Reports the ambiguity as "fatal" and quits. These are cases where no reasonable resolution of the ambiguity is available. Such situations can only arise in "complex" languages.

From the input described in Table II, Rword would produce a processor which would itemize a character string in the manner indicated and would call the user-written execute procedure "SWITCH1" when item "4" (. . .) was encountered.

The purpose of this procedure would be to switch to a processing scheme appropriate for handling remarks. In AED-0 "remarks" are comments within a statement and are delineated by ". . ." and "/ /" or "$,". The remark

TABLE III. RWORD Input

Class descriptions	*Meaning*
BEGIN	
ALLBUT = $01, 02, 03, 04, 05, 06, 07, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 23, 24, 25, 26, 27, 30, 31, 32, 33, 34, 35, 36, 37, 40, 41, 42, 43, 44, 45, 46, 47, 50, 51, 52, 54, 55, 56, 57, 60, 62, 63, 64, 65, 66, 67, 70, 71, 72, 73, 74, 75, 76, 77$	All characters but "$"(53) and "/"(61) are in the class ALLBUT.
END	
Item definitions	***Meaning***
BEGIN	
IGNORE = ALLBUT $,	All characters of the class "ALLBUT" are to act as ignorable delimiter.
NOEND(1) = $ U '/$,	A "$" or a "/" by themselves do not end the remark.
TERMIN(3, SWITCH2) = $/', U '//'/$,	A "$," or "//" will result in a call on SWITCH2.
END FINI	

Note. The apostrophe is used in item definition NOEND and TERMIN so that characters / and , will not be treated as operators in the description language.

processor should read all the characters of the remark and switch back (by a call to execute procedure SWITCH2) to the normal processing procedure upon encountering a "$," or "//". The RWORD input in Table III would produce the desired result.

In actual practice the two processors are combined at an intermediate stage by RWORD, and the user would end up with one lexical processor which would incorporate both processing schemes.

Processor Functioning

The RWORD system is of considerable interest in its own right, since it directly applies a portion of the abstract theory of computing machines in an entirely different field. The general approach used in RWORD is based on the theory of finite-state automata or sequential machines. Finite automata are machines having only a finite number of internal states that can be used for memory and computation (as opposed to Turing machines). The specialized lexical processors which RWORD produces are computer plex[1] structures which function as finite-state machines. At any point in the reading of the input character string this "machine" will be in some unique state. Upon reading the next character the machine will change to a new state as dictated by the particular character read. Obviously, a machine which does nothing but read characters and change states is not of much value. However, these machines provide the structure upon which the actual processors are built. As an example of a finite-state machine, the lexical language defined by the regular expressions

```
BEGIN
S(1) = B/A$,
S(2) = A$,
S(3) = A/C$,
END FINI
```

would result in a three-state finite-state "machine," represented by Figure 1.

The circles or nodes are the *states* and the arrows are the *transitions*. Each state represents a unique *condition-of-tension* in reading the character string, and all such legal possible conditions are represented. At each state the machine reads a new character and changes to the new state indicated by the transition. If the reading operation comes up with any character other than those indicated on the transitions, an error condition is indicated. Thus the character string "BB" would cause an error condition. The first B would take us to node 3, but node 3 has no transition for the second B.

To make this "machine" function as a processor, various operations must be carried out each time the machine changes state, i.e. on each transition. These operations consist of the atomic actions of lexical processing. The "actions" associated with the transitions of our three-state machine are shown in Figure 2.

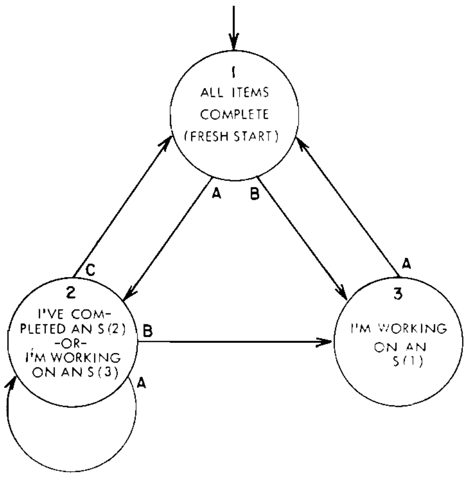


FIG. 1

Here the "action" of "report S(3)" means report all the characters currently in the character buffer as an S(3) and clear the buffer.

This diagram or graph of the finite-state machine (Figure 2) represents the complete functioning processor. The processor would break up a character string in the manner indicated by the input expressions, reporting items as they were recognized.

It can be seen that the order of execution of the various "actions" is important. On the "C" transition out of node 2 we add the character to the buffer before reporting the item, giving us an item (3) "AC" and an empty buffer. On the "B" transition from the same node we report the item before adding the character to the buffer, giving us an item (2) "A" and a "B" in the buffer. Each atomic

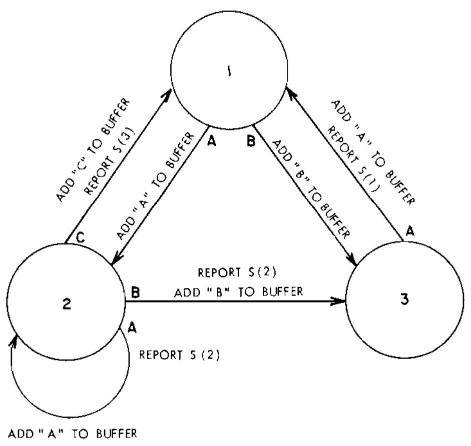


FIG. 2

[1] AED-0 Programming Manual Preliminary Release #2, D. T. Ross, Oct.-Dec., 1964. "The term plex was coined early in the Computer-Aided Design Project as a derivative from the word *plexus*, which has the dictionary meaning 'an interwoven combination of parts in a structure; a network.' Early usage of the term stressed primarily the data structure aspect of the concept, although reference always was made to algorithms which would interpret or give meaning to the data structure." (See [10].)

action of lexical processing is in the form of a subroutine, and the appropriate sequence of calls made on each transition. The "read another character" action is the last action on each transition. Some other "actions" are:

"Report an error condition"
"Call a user-designated subroutine"

When the user has employed the counting capability in his regular expressions (as in SYM(1) = LET↑/(LET ∪ DIG)*5 $,) the actions required are:

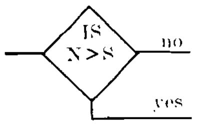
"Increment the counter"
"Reset the counter"
"Test the counter"

For some languages it cannot be determined whether a character belongs to the current item or begins a new item until several more characters have been read. The processor does not, however, back up or reread. Complex languages (those requiring "look ahead") use the action "Mark a possible end of an item" and the reporting of items no longer consists of emptying the character buffer, but rather involves reporting the characters up to the appropriate item-ending mark.

Although Rword handles involved languages with ease, the diagrams representing their finite-state machines soon become extremely interwoven and difficult to follow. Figure 3 shows the diagram of the finite-state machine for the relatively simple language described in Table 2 as an example of Rword input.

To conserve space the actions are abbreviated as follows:

$\boxed{\text{IN}}$	Put the character in the character buffer
$\boxed{5}$	Report all the characters in the buffer as item 5, empty buffer
$\boxed{\text{SWITCH1}}$	Call procedure "SWITCH1"
$\boxed{N=0}$	Reset the counter
$\boxed{N=N+1}$	Increment the counter



Test the counter

Note that the characters of ignorable items are not put in the character buffer, and the item is not reported.

Remember that the action "read another character" occurs as the last action on every transition.

Construction of the RWORD System

The general construction of the Rword System is shown in Figure 4.

The portion of the system which automatically generates lexical processors is divided into two parts. In the first part the regular expressions defining the lexical types which make up a user language are accepted as input. This first phase generates an AED-0 procedure which is in the form of an ordered sequence of calls on a library of standard procedures used to build an information table. The information table guides the operation of the lexical processor.

In the second phase of the Rword system, the AED-0 procedure is compiled and loaded along with the package of table-building procedures. The output of this phase is a macroinstruction version of the lexical processor table. If several lexical processing schemes are to be combined, each of the compiled AED-0 procedures is loaded at the

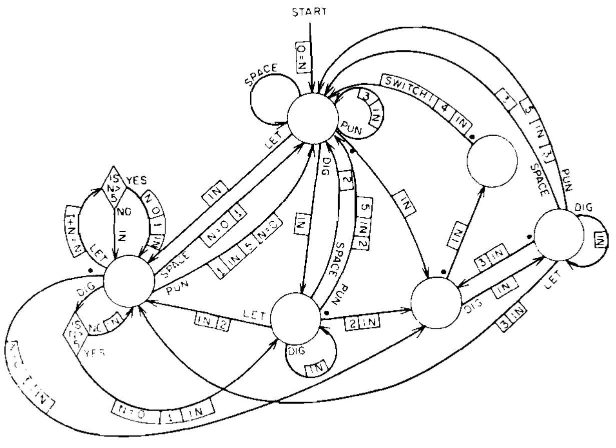


Fig. 3

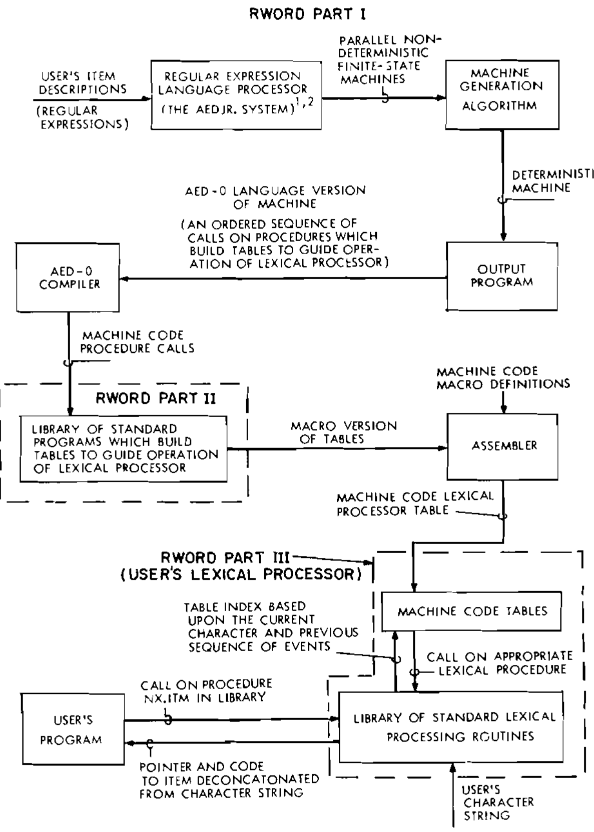

RWORD PART I
USER'S ITEM DESCRIPTIONS (REGULAR EXPRESSIONS)
REGULAR EXPRESSION LANGUAGE PROCESSOR (THE AEDJR. SYSTEM)1,2
PARALLEL NON-DETERMINISTIC FINITE-STATE MACHINES
MACHINE GENERATION ALGORITHM
DETERMINISTIC MACHINE
AED-0 LANGUAGE VERSION OF MACHINE
(AN ORDERED SEQUENCE OF CALLS ON PROCEDURES WHICH BUILD TABLES TO GUIDE OPERATION OF LEXICAL PROCESSOR)
AED-0 COMPILER
OUTPUT PROGRAM
MACHINE CODE PROCEDURE CALLS
RWORD PART II
LIBRARY OF STANDARD PROGRAMS WHICH BUILD TABLES TO GUIDE OPERATION OF LEXICAL PROCESSOR
MACRO VERSION OF TABLES
MACHINE CODE MACRO DEFINITIONS
ASSEMBLER
MACHINE CODE LEXICAL PROCESSOR TABLE
RWORD PART III (USER'S LEXICAL PROCESSOR)
TABLE INDEX BASED UPON THE CURRENT CHARACTER AND PREVIOUS SEQUENCE OF EVENTS
MACHINE CODE TABLES
CALL ON APPROPRIATE LEXICAL PROCEDURE
CALL ON PROCEDURE NX.ITM IN LIBRARY
USER'S PROGRAM
LIBRARY OF STANDARD LEXICAL PROCESSING ROUTINES
POINTER AND CODE TO ITEM DECONCATONATED FROM CHARACTER STRING
USER'S CHARACTER STRING

Fig. 4

start of phase II with the table-building procedures. In the second part a macroversion of the combined lexical processing activities is then generated.

The macroversion of the table is assembled with a macroassembler (the Fap macroassembler for the IBM 7094) to form a machine code version of the information table. The information table is loaded together with a package of lexical processing procedures to form the lexical processor.

The third phase of the system is the operation of the user's lexical processor. A call is made on the procedure NX.ITM from within a user program. NX.ITM is one of the lexical processing procedures. NX.ITM reads a character from a preestablished file which contains the user's character string and, based upon the character read, looks in the appropriate spot in the information table. The information provided in the table includes a pointer to the appropriate lexical processing procedures and the arguments the procedure requires to carry out its function.

Form of the Processors

Rword-generated lexical processors have two logical parts:

1. The run-time package containing the subroutines representing the atomic lexical "actions" and the user-written subroutines.
2. A plex structure which forms the finite-state machine with the appropriate calls on the action routines.

Each state of the finite-state machine is in the form of a table with an entry for each character. The entries consist of transfers to other states along with calls on the appropriate action subroutines.

The actual output of Rword (the finite-state machine) is a sequence of macro calls which, when assembled with the proper macro definitions, yields the desired plex structure. This form was chosen to provide a degree of machine independence to the Rword output.

Constructing the Processors: Phase I

One of the advantages of employing the principles of finite-state machines is that a straightforward technique exists for the construction of these machines from the regular expressions [7–9]. The following is an outline of the somewhat simplified technique we employed.

The first step in the process consists of constructing a simpler form of finite-state machine for each of the expressions. For example, the expressions from our first example yield the machines as shown in Figure 5.

It can be seen that these diagrams correspond exactly to the item descriptions in that one can move from the input state to the output state of each diagram with, and only with, character strings that satisfy its item description. Because of this exact correspondence, the process of construction is reasonably straightforward.

Although we have spoken of these diagrams as representing separate "machines" for each expression, each begins at "INPUT" and ends at "OUTPUT"; so they actually represent only one, large "machine." This "machine" cannot serve, however, as the basis of a processor since there are states which transfer to more than one other state on the same character or class. Since it cannot be determined which state the "machine" should transfer to on a given character, the "machine" is called a *nondeterministic machine*. (Note, for example, that state 2 transfers to both the output state and state 2 on a "LET".)

The technique we employed to develop the *deterministic* machine, which will be the basis of our output processor, is briefly as follows.

Whenever a state transfers to more than one other state on a given character, we represent this indecision by creating a single, compound state having the combined properties of the states transferred to. Thus, since the input state transfers to both state 4 and state 6 on a dot, we create the compound state "4, 6". This compound state combines the properties of states 4 and 6 (i.e. it transfers to state 5 or OUTPUT on a "DIG" and to state 7 on a dot). When this process is continued systematically to completion, we have a "machine" in which each state transfers to a unique state on each character. This is the deterministic, finite-state machine, which is our objective.

The technique for creating compound states does not always resolve indeterminacies. For instance, if we have a state which transfers to the compound state 3, 4 and also to state 3, we are in trouble, since the combining process simply yields the compound state 3, 4 *again*, and we have not resolved the indeterminacy. We have only made an arbitrary decision. Such situations arise from

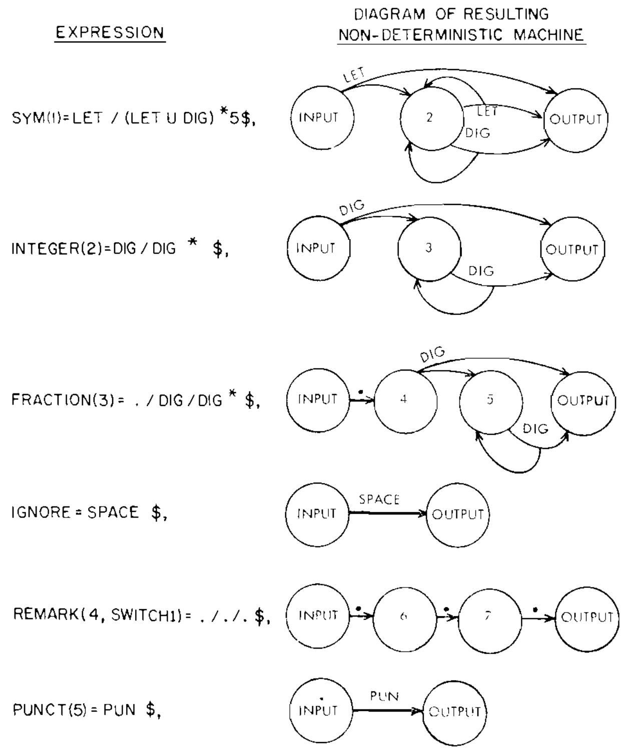

EXPRESSION
DIAGRAM OF RESULTING NON-DETERMINISTIC MACHINE
SYM(1)=LET / (LET U DIG) *5$,
INPUT
2
LET
LET
DIG
OUTPUT
INTEGER(2)=DIG / DIG * $,
DIG
3
DIG
FRACTION(3)= . / DIG / DIG * $,
4
5
DIG
DIG
IGNORE = SPACE $,
SPACE
REMARK(4, SWITCH1)= . / . / . $,
6
7
PUNCT(5) = PUN $,
PUN

Fig. 5

ambiguities in the input language. In general it can be shown that whenever two nonidentical transitions being combined have any simple states in common, the input language is ambiguous. As indicated previously, Rword handles these ambiguities in different ways depending on their nature and context.

Rword builds the nondeterministic machine as a plex in the computer (see Figure 5), but the deterministic machine with its associated actions is generated as an AED-0 program. This program consists entirely of calls with appropriate arguments on various subroutines, each call corresponding to a single transition of the processor. These routine calls and arguments are such that the program could be made to perform the lexical processing by incorporating the appropriate atomic actions in the subroutines. Although this processor could perform the specified task, it would be quite inefficient both in speed and in storage requirements. Clearly, some optimizing is needed.

Constructing the Processors: Phase II

In actual practice, the AED-0 program is an intermediate output of Rword phase I. Phase II of Rword has two functions:

1. It alters the form of the processor to provide maximum processing speed and minimum storage requirements.
2. When more than one lexical scheme is required in the processor (for context-dependent languages), it combines the several schemes (each represented by a phase I output) into a single processor.

As indicated previously, Rword's final output is a sequence of macro calls. These calls are generated in phase II by compiling and executing the program (or programs) of phase I. Thus the actual function of the subroutines called in that program is not that of lexical processing but rather it consists of assembling the appropriate macro calls for the particular transition of the processor. The macro definitions are all that must be altered when changing hardware configurations.

The final step consists of assembling the phase II output and combining it with the Rword run-time package to form the final lexical processor.

Future Developments and Applications

The AED Rword system has been operational since late 1966, and many of its features resulted from suggestions made by early users. This process of responding to user needs has continued to increase the usefulness of the system.

In our view, a major objection to the original Rword system was its limited functional design as a character processor. In fact, the very use of the term "character" implies an improperly restrictive application of the system. The only property of characters that is essential to the operation of Rword is that they be available to the processor in an ordered sequence. Thus the techniques of lexical analysis can be applied to any elements which have this property. It is more profitable to think of the generalized input to the Rword processors as "events" and to describe these processors as generalized decision-making mechanisms with an input sequence of "events" and with the output being the result of decisions based on the ordering of the events.

One application which comes to mind is in the later stages of a compiler where the input would consist of types of source language statements and the output would be blocks of object code appropriate to a particular sequence of statement types. The technique might also find application in the area of theory-of-games or as a portion of a management system.

We feel that techniques of finite-state machines provide the basis of a very powerful and general tool which should find application in fields far removed from computing machinery, and hope that the Rword system might play a useful role in this development.

Postscript

The AED Rword system had its beginning as a classroom exercise during the orientation period for new members of the AED Cooperative Program in the spring of 1966. As a result many other members of the project played a role in various parts of its design and execution.

In the time since this paper was drafted for joint authorship perusal, the original AED Rword system has undergone several important changes by other staff members of the M.I.T. Computer-Aided Design Project. The system now is character-set independent, more foolproof and efficient in operation, and the run-time package has been prepared for the IBM 360 and Univac 1108 computers as well as the IBM 7094. Many applications of the resulting improved system have been made, and Rword has played a crucial role in the bootstrapping of the various AED systems to the above computers. Rword I and II are being prepared for bootstrapping, and although the use of Rword-generated finite-state machines in AED compiler code generation has not yet been initiated, such machines are now being used in the new language independent Print Algorithm (PRALG) of AED for generating highly formatted source language layouts for readable program documentation.

The following people have participated in various ways, as helpful users, extenders, or question answerers, in the continuing Rword development: R. J. Bigelow, C. G. Feldmann, P. Johansen, P. T. Ladd, G. L. Lane, R. B. Lapin, J. E. Rodriguez, J. F. Walsh, and B. L. Wolman.

Received April, 1968

REFERENCES

1. Ross, D. T. The AED approach to generalized computer-aided design. Proc. ACM 22nd Nat. Conf., 1967, pp. 367–385.
2. ——. An algorithmic theory of language. Rep. No. ESL-TM-156, Electron. Syst. Lab., MIT, Cambridge, Mass., Nov. 1962.

3 Cheatham, T. E., Jr. The TGS-II translator generator system. Proc. IFIP Cong., 1965, Vol. 2, pp. 592-593.

4. Conway, M. E. Design of a separable transition-diagram compiler. *Comm. ACM 6*, 7 (July 1963), 396-408.

5. Floyd, R. W. The syntax of programming language—a survey. *IEEE Trans.* (Aug. 1964), 346-353.

6. Ledley, R. S., and Wilson, J. B. Automatic-programming-language translation through syntactical analysis. *Comm. ACM 5*, 3 (Mar. 1962), 145-155.

7. McNaughton, R. Techniques for manipulating regular expressions. MIT memo, Cambridge, Mass., Nov. 1965.

8. ——, and Yamada, H. Regular expressions and state graphs for automata. In Moore, E. F. (Ed.), Sequential machines—selected papers, Bell Tel. Labs., Inc., Murray Hill, N.J.

9. Shannon, D. E., and McCarthy, J. *Automata Studies*. Princeton U. Press, Princeton, N. J., 1956.

10. Ross, D. T. A generalized technique for symbol manipulation and numerical calculation. *Comm. ACM 4*, 3 (Mar. 1961), 147-150.

11. Feldman, J., and Gries, D. Translator writing systems. *Comm. ACM 2*, 2 (Feb. 1968), 77-113.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Electronic Systems Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP --

3. REPORT TITLE

Computer-Aided Design for Numerically Controlled Production

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Final Technical Report for period 1 May 1967 through 30 January 1970

5. AUTHOR(S) *(Last name, first name, initial)*
Ward, John E.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1970	121	

8a. CONTRACT OR GRANT NO. F33615-67-C-1531 F33615-69-C-1341	9a. ORIGINATOR'S REPORT NUMBER(S) M.I.T. Report ESL-FR-420
b. PROJECT NO. 863-7	
c. 86309	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* AFML-TR-70-78
d.	

10. AVAILABILITY/LIMITATION NOTICES

This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of the Air Materials Laboratory, Wright-Patterson Air Force Base, Ohio 45433

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Air Force Materials Laboratory, Wright-Patterson Air Force Base, Ohio 45433

13. ABSTRACT This report summarizes the activities of the M.I.T. Computer-Aided Design (CAD) Project from 1 May 1967 through 30 January 1970 in the final implementation phase of a generalized "system of software systems" for generating specialized problem-oriented man-machine problem-solving systems. Known as the AED approach (for Automated Engineering Design) the Project results are applicable not only to mechanical design, but to arbitrary scientific, engineering, management, and production system problems as well. Program accomplishments are supported by hardware and software innovations in computer graphics. All results have been programmed using machine-independent techniques in tke Project's AED-0 Language, based on Algol-60.

During this concluding 32-month phase of the 10-year CAD program at M.I.T., the major emphasis has been on bootstrapping the AED systems to third-generation computers. A series of field-trial systems was made available to industry, culminating in July 1969 in formal release of Version 3 of AED for IBM 360-series computers in both batch and time sharing, and partial completion of a compatible version for the Univac 1108 computer. Report topics include: The bootstrapping process; user documentation for the AED system; several application studies to demonstrate use of AED techniques in language design, system building, and computer graphics; and Project interaction with industry.

14. KEY WORDS

programming languages	computer-aided design	display systems
compilers	AED	
software	digital computers	computer graphics

DD FORM 1 NOV 65 1473 (M.I.T.)

UNCLASSIFIED
Security Classification

REPORTS DISTRIBUTED BY THE COMPUTER-AIDED DESIGN PROJECT

REPORT AND TECHNICAL MEMO NUMBERS	DDC NOS.	TITLE	AUTHOR(S)	DATE
8436-TM-1	AD 243 156 PB 155 406	Papers on the APT Language	Ross, D.T. Feldmann, C.G.	6/60
8436-TM-2	AD 248 436 PB 155 407	Method for Computer Visualization	Smith, A.F.	9/60
8436-TM-3	AD 248 437 PB 155 408	A Digital Computer Representation of the Linear, Constant-Parameter Electric Network	Meyer, C.S.	8/60
8436-TM-4	AD 252 060 PB 155 409	Computer-Aided Design: A Statement of Objectives	Ross, D.T.	9/60
8436-TM-5	AD 252 061 PB 155 410	Computer-Aided Design Related to the Engineering Design Process	Coons, S.A. Mann, R.S.	10/60
8436-R-1	AD 253 676 PB 155 553	Automatic Feedrate Setting in Numerically Controlled Contour Milling	Welch, J.D.	12/60
8436-IR-1	AD 252 062 PB 155 405	Investigations in Computer-Aided Design Interim Report No. 1	Project Staff	1/61
8436-IR-2	AD 269 573	Investigations in Computer-Aided Design Interim Report No. 2	Ross, D.T. Coons, S.A.	11/61
ESL-R-132	AD 274 985	Design of a Remote Display Console	Randa, G.C.	2/62
ESL-IR-138	AD 282 679	Investigations in Computer-Aided Design for Numerically Controlled Production--Interim Technical Progress Report No. 3 and 4 ASD-TR-7-820 (IR 3 and 4)	Ross D.T. Coons, S.A.	5/62
ESL-TM-156	AD 296 998	An Algorithmic Theory of Language	Ross, D.T.	11/62
Lincoln Lab Technical Report No. 296	AD 464 549	Sketchpad: A Man-Machine Graphical Communication System	Sutherland, I.E.	1/63
ESL-TM-164	AD 403 685	Investigations in Computer-Aided Design for Numerically Controlled Production--Interim Technical Progress Report No. 5 ASD-TR-7-820 (IR 5)	Ross, D.T. Coons, S.A.	2/63
ESL-TM-167	AD 406 608	Specialized Computer Equipment for Generation and Display of Three Dimensional Curvilinear Figures	Stotz, R.H.	3/63
ESL-TM-169	AD 404 832	An Outline of the Requirements for a Computer-Aided Design System	Coons, S.A.	3/63
ESL-TM-170	AD 405 882	Theoretical Foundations for the Computer-Aided Design System	Ross. D.T. Rodriguez, J.E.	3/63
ESL-TM-173	AD 406 855	Sketchpad III, Three Dimensional Graphical Communication with a Digital Computer	Johnson, T.E.	5/63
ESL-IR-180	AD 418 183	Investigations in Computer-Aided Design for Numerically Controlled Production--Interim Technical Progress Report No. 6 ASD-TR-7-820 (IR 6)	Ross, D.T. Coons, S.A.	6/64
ESL-IR-202	AD 442 880	Investigations in Computer-Aided Design for Numerically Controlled Production--Interim Technical Progress Report No. 7 ASD-TR-7-820 (IR 7)	Ross. D.T. Coons, S.A.	6/64
ESL-TM-211	AD 453 881	AED Jr.: An Experimental Language Processor	Ross, D.T.	9/64
ESL-TM-212	AD 453 880	Implications of Computer-Aided Design for Numerically Controlled Production	Ross, D.T.	9/64
ESL-TM-220	AD 472 147	Some Experiments with an Algorithmic Graphical Language	Lang, C. A. Polansky, R. B. Ross, D. T.	8/65
ESL-IR-221	AD 604 678	Investigations in Computer-Aided Design for Numerically Controlled Production -- Interim Engineering Progress Report IR 8-236-I	Ross, D. T. Coons, S. A. Ward, J. E.	12/65
ESL-TM-228	AD 461 412	An Approach to Computer-Aided Preliminary Ship Design	Hamilton, M. L. Weiss, A. D.	1/65
ESL-IR-241	AD 467 764	Investigations in Computer-Aided Design for Numerically Controlled Production -- Interim Engineering Progress Report IR 8-236-II	Ross, D. T. Coons, S. A. Ward, J. E.	6/65

REPORTS DISTRIBUTED BY THE COMPUTER-AIDED DESIGN PROJECT

REPORT AND TECHNICAL MEMO NUMBERS	DDC NOS.	TITLE	AUTHOR(S)	DATE
ESL-IR-262	AD 482837	Investigations in Computer-Aided Design for Numerically Controlled Production -- Interim Engineering Progress Report IR 8-236-III	Ross, D. T. Coons, S. A. Ward, J. E.	3/66
ESL-IR-278	AD802213	Investigations in Computer-Aided Design for Numerically Controlled Production -- Interim Engineering Progress Report IR 8-236-IV and V	Ross, D. T. Coons, S. A. Ward, J. E.	8/66
ESL-R-305	AD 814912	The AED Approach to Generalized Computer-Aided Design	Ross, D. T.	4/67
ESL-R-306	AD 815395	Translation Between Artificial Programming Languages	Lapin, R. B.	4/67
ESL-IR-320	AD 821385	Investigations in Computer-Aided Design for Numerically Controlled Production -- Interim Engineering Progress Report IR 8-236-VI	Ross, D. T. Ward, J. E.	8/67
ESL-FR-351	AD 850036	Investigations in Computer-Aided Design for Numerically Controlled Production -- Final Technical Report for period 1 December 1959 - 3 May 1967. (Also Air Force Report AFML-TR-68-206)	Ross, D. T. Ward, J. E.	5/68
ESL-R-356	AD 685202	An Integrated Hardware-Software System for Computer Graphics in Time-Sharing (Also Project MAC Technical Report MAC-TR-56)	Thornhill, D. E. Stotz, R. H. Ross, D. T. Ward, J. E.	12/68
ESL-TM-394	AD 859084	The SHOWIT System: An Example of the Use of the AED Approach	Ross, J. R. Ross, D. T.	6/69
ESL-R-397	AD 860624	Interactive Syntax Definition Facility	Eanes, R. S.	9/69
ESL-R-398	AD 697759	A Graph Model for Parallel Computations	Rodriguez, J. E.	9/69
ESL-R-405	AD 863500	Introduction to Software Engineering with the AED-0 Language	Ross, D. T.	10/69
ESL-R-406	AD 866123	AED Programmer's Guide	Feldmann, C. G. (Editor) Ross, D. T. Rodriguez, J. E.	11/69
ESL-FR-420		Computer-Aided Design for Numerically Controlled Production -- Final Technical Report for period 1 May 1967 - 30 January 1970. (Also Air Force Report AFML-TR-70-78)	J. E. Ward	6/70